

**Н. Ф. Хайрова,
О. Ж. Мамырбаев,
С. В. Петрасова,
К. Ж. Мухсина**

СОВРЕМЕННЫЕ ТЕХНОЛОГИИ ОБРАБОТКИ ТЕКСТОВЫХ ДАННЫХ НА БАЗЕ ПАКЕТА NLTK PYTHON



МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ УКРАИНЫ

НАЦИОНАЛЬНЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ХАРЬКОВСКИЙ ПОЛИТЕХНИЧЕСКИЙ ИНСТИТУТ»

Н. Ф. Хайрова, О. Ж. Мамырбаев, С. В. Петрасова, К. Ж. Мухсина

**СОВРЕМЕННЫЕ ТЕХНОЛОГИИ ОБРАБОТКИ ТЕКСТОВЫХ ДАННЫХ
НА БАЗЕ ПАКЕТА NLTK PYTHON**

Учебное пособие

по курсу «Современные технологии обработки текстовых данных»
для студентов специальности «Прикладная и компьютерная лингвистика»

Рекомендовано Ученым советом НТУ «ХПИ»

и

Ученым советом РГП «Институт информационных и вычислительных
технологий» Комитета науки Министерство образования и науки Республики
Казахстан

Харьков

НТУ «ХПИ»

2020

УДК 004.9(075.8)

C56

Рецензенты:

И. В. Шостак, д-р. техн. наук, профессор, Национальный аэрокосмический университет им. Н. Е. Жуковского «Харьковский авиационный институт»;

А. Л. Ерохин, д-р. техн. наук, профессор, Харьковский национальный университет радиоэлектроники

*Рекомендовано Ученым советом РГП «Институт информационных и вычислительных технологий» Комитета науки Министерство образования и науки Республики Казахстан
протокол № 6 от 28 мая 2020 г. и*

*Ученым советом Национального технического университета «Харьковский
политехнический институт» протокол № 5 от 13 октября 2020 г.
как учебное пособие для студентов специальности
«Прикладная и компьютерная лингвистика»*

У навчальному посібнику наведено матеріал, що охоплює широке коло питань, пов'язаних з використанням пакета NLTK у Python при вирішенні завдань комп'ютерної лінгвістики. Матеріал проілюстровано наочними практичними прикладами, розділи включають практичні завдання та контрольні запитання.

Призначено для студентів спеціальності «Прикладна та комп'ютерна лінгвістика».

Хайрова Н. Ф., Мамырбаев О. Ж., Петрасова С. В., Мухсина К. Ж.

**C56 Современные технологии обработки текстовых данных на базе
пакета NLTK Python : учеб. пособ. / Н. Ф. Хайрова, О. Ж. Мамырбаев,
С. В. Петрасова, К. Ж. Мухсина. Харьков : НТУ «ХПИ», 2020. 134 с.
На русском языке.**

ISBN 978-617-7305-46-9

В учебном пособии приведен материал, охватывающий широкий круг вопросов, связанных с использованием пакета NLTK в Python при решении задач компьютерной лингвистики. Материал проиллюстрирован наглядными практическими примерами, разделы включают практические задания и контрольные вопросы.

Предназначено для студентов специальности «Прикладная и компьютерная лингвистика».

Ил. 20. Табл. 5. Библиогр. 10 назв.

УДК 004.9(075.8)

ISBN 978-617-7305-46-9

© Н. Ф. Хайрова, О. Ж. Мамырбаев,
С. В. Петрасова, К. Ж. Мухсина, 2020

СОДЕРЖАНИЕ

ВВЕДЕНИЕ	7
Тема 1. ВВЕДЕНИЕ В NATURAL LANGUAGE PROCESSING	8
1.1. Задачи и приложения NLP	8
1.2. Установка открытой библиотеки NLTK	12
Контрольные вопросы и практические задания к теме “Введение в Natural Language Processing”	15
Тема 2. СТРОКИ PYTHON	16
2.1. Особенности работы с текстовыми строками в Python.....	16
2.2. Форматирование строк Python.....	18
2.3. Строки в формате Unicode в Python	21
2.4. Строковые методы и функции	22
Контрольные вопросы и практические задания по теме «Строки Python»	25
Тема 3. РЕГУЛЯРНЫЕ ВЫРАЖЕНИЯ	29
3.1. Введение в регулярные выражения.....	29
3.2. Использование регулярных выражений в Python	33
3.3. «Нежадные» квантификаторы в Python	35
3.4. Круглые скобки в регулярных выражениях Python.....	35
3.5. Методы и функции модуля re	39
Контрольные вопросы и практические задания по теме «Регулярные выражения»	45
Тема 4. НОРМАЛИЗАЦИЯ ТЕКСТА	49
4.1. Сегментация слов	49
4.2. Нормализация	51

4.3. Стемминг	52
4.4. Сегментация предложений в тексте	54
4.5. Токенизация в NLTK Python	55
4.6. Использование регулярных выражений для стемминга в Python	57
4.7. Встроенные стеммеры пакета NLTK	59
4.8. Лемматизатор пакета NLTK	61
Контрольные вопросы и практические задания по теме «Нормализация текста»	62
Тема 5. POS TAGGING (МОРФОЛОГИЧЕСКАЯ РАЗМЕТКА)	64
5.1. Задачи POS-тегирования	64
5.2. Автоматическое тегирование средствами пакета nltk. Функция pos_tag()	68
5.3. Установка пакета rymorphy2	72
5.4. Rymorphy2 – морфологический анализатор для русского и украинского языков	72
Контрольные вопросы и практические задания по теме «POS Tagging (морфологическая разметка)»	78
Тема 6. СИНТАКСИЧЕСКИЙ АНАЛИЗ	80
6.1. Сущность синтаксического анализа текста	80
6.2. Деревья синтаксической зависимости	81
6.3. Проективные деревья синтаксической зависимости	84
6.4. Система составляющих	86
6.5. Использование контекстно-свободных грамматик для реализации размеченных систем составляющих	89
Контрольные вопросы и практические задания по теме «Строки Python»	91

Тема 7. СЕМАНТИЧЕСКАЯ ОБРАБОТКА. РАБОТА С ТЕЗАУРУСОМ WORDNET	93
7.1. Общее описание лингвистического ресурса WordNet.	93
7.2. Работа со словарями WordNet с помощью NLTK	94
7.3. Иерархия семантических отношений WordNet.....	97
7.4. Использование WordNet для определения семантической близости слов.....	102
Контрольные вопросы и практические задания по теме «Семантическая обработка. Работа с тезаурусом WordNet»	104
Тема 8. РАБОТА С КОРПУСАМИ В NLTK	106
8.1. Работа с модулем nltk.corpus пакета NLTK.....	106
8.1.1. Корпус Проекта Гутенберга.....	108
8.1.2. Корпус webtext.....	110
8.1.3. Корпус nps_chat	111
8.1.4. Корпус Брауна (Brown corpus).....	112
8.1.5. Reuters-корпус	113
8.1.6. Корпус inaugural	113
8.2. Текстовые корпуса на различных языках. Использование кодировок.....	114
8.3. Загрузка собственного корпуса	117
8.4. Работа с размеченными корпусами в NLTK	117
Контрольные вопросы и практические задания по теме «Работа с корпусами в nltk».....	120
Тема 9. ПРОСТЫЕ СТАТИСТИЧЕСКИЕ ИССЛЕДОВАНИЯ КОРПУСОВ	122
9.1. Создание объекта nltk.Text.....	122
9.2. Использование пакета Matplotlib для построения графиков и диаграмм.....	124

9.3. Построение частотного распределения слов в корпусе	125
9.4. Коллокации и биграммы (collocations and bigrams) в тексте	130
Контрольные вопросы и практические задания по теме «Простые статистические исследования корпусов».....	131
СПИСОК ЛИТЕРАТУРЫ.....	133

ВВЕДЕНИЕ

Сегодня технологии **Natural Language Processing (NLP)** становятся все более распространенными. Например, телефоны и планшеты поддерживают интеллектуальный предиктивный ввод текста и распознавание рукописного текста; системы информационного поиска анализируют неструктурированный текст; системы машинного перевода позволяют извлекать текст китайского языка и затем транслировать его на украинском, русском или английском языках. Последние пять лет чат-боты начали доминировать в предпринимательстве и рекламе, а Sentiment analysis позволяет определять настроения в социальных сетях, твитах и блогах. Предоставляя более естественные человеко-машинные интерфейсы и более сложный доступ к хранимой информации, языковая обработка стала занимать одно из центральных мест в многоязычном информационном обществе.

Данное учебное пособие, предоставляя доступное введение в область NLP, рассматривает такие традиционные подходы обработки текста, как токенизация, нормализация, стемминг, POS-tagging, синтаксический parser, семантический анализ, с подключением к WordNet, регулярные выражения. Пособие также включает описание возможностей практической реализации NLP задач инструментами библиотеки NLTK Python.

Natural Language Toolkit (NLTK) представляет собой открытую библиотеку, включающую обширное программное обеспечение, данные и документацию, которые можно бесплатно загрузить с сайта <http://nltk.org/>. В настоящее время NLTK считается хорошим инструментом для обучения и работы в вычислительной и компьютерной лингвистике (“computational linguistics”)

Пособие может быть использовано для изучения дисциплин, связанных с обработкой естественного языка, компьютерной лингвистикой, интеллектуальным анализом текста, а также для индивидуального изучения.

Тема 1. ВВЕДЕНИЕ В NATURAL LANGUAGE PROCESSING

1.1. Задачи и приложения NLP.

1.2. Установка открытой библиотеки NLTK.

Контрольные вопросы и практические задания к теме “Введение в Natural Language Processing”.

1.1. Задачи и приложения NLP

Natural Language Processing (NLP) представляет собой активно применяемую на практике область исследований, целью которой является обработка языка на базе лингвистических, статистических и информационных технологий. Область исследований NLP частично пересекается с компьютерной и математической лингвистикой. Однако, основной задачей компьютерной лингвистики является моделирование процесса понимания смысла текстов, включающее анализ – переход от текста к формализованному представлению его смысла, и синтез – переход от формализованного представления смысла к текстам естественного языка. Для решения такого рода задачи необходимо моделировать функции человеческого интеллекта, связанные с использованием естественного языка, такие, как реферирование, перевод, извлечение и идентификация знаний, ответы на общие и специальные вопросы, перифраз и другие функции, связанные с пониманием текста или речи.

Сегодня направление NLP, под которым подразумевают любой тип компьютерной обработки естественного языка – от подсчета числа слов до сравнения различных стилей текста и «понимания» смысла текста, – имеет более практическую направленность, чем компьютерная лингвистика в целом. Подходы NLP, часто называемые “Computational Linguistics”, становятся одним из наиболее интенсивно развивающихся направлений информационных технологий. Причина интереса к подходам и задачам обработки текстовой информации во многом связана с непрерывным увеличением объема текстовых данных в социальных и интернет сетях.

К наиболее известным приложениям NLP относятся:

- вопросно-ответные системы (Question answering (QA));

- системы извлечения информации и извлечение фактов (Information extraction (IE));
- автоматический анализ мнений (Sentiment analysis или Opinion Mining);
- системы машинного перевода (Machine translation (MT));
- автоматическое перефразирование, рерайтинг, синонимайзер (Paraphrase);
- системы автоматического реферирования (Text Summarization);
- человеко-машинный диалог в интерфейсах систем (Human-Machine Dialogue), включающий Chatbots;
- системы информационного поиска (Information retrieval);
- автоматическая классификация и кластеризация текстов (Text classification);
- автоматическая проверка орфографии (Spelling Correction);
- автоматическая генерация онтологии (Ontology generation);
- автоматическое определение тематики текста (Topic identification) и другие приложения.

Практическая реализация перечисленных приложений NLP широко варьируется от направлений, по которым есть коммерческие продукты и системы, до направлений, в реализации которых задействованы работающие алгоритмы, и направлений, существующих только в экспериментальных разработках.

К направлениям обработки текста, по которым существуют хорошо реализованные коммерческие предложения, можно отнести приложения по определению спама в получаемом пользователем потоке e-mail текстов. И хотя эти приложения продолжают принимать достаточное количество ошибочных решений, они включены в современные почтовые сервисы.

Еще одним примером коммерциализации является синтез речи (speech synthesis), имеющий успешно разработанные приложения, подобно спецификации SALT (Speech Application Language Tags).

Примером не доведенных до коммерческой реализации задач NLP, но имеющих работающие алгоритмы, является задача извлечения информации из текста (**Information Extraction**). Приложения, связанные с извлечением информации,

которые извлекают из текстов мнения автора (положительные или отрицательные) о том или ином предмете, представляют задачи **Opinion Mining**. Данный процесс называется также *sentiment analysis*, *opinion extraction* или *opinion mining*, *sentiment mining*, *subjectivity analysis* и имеет некоторые другие наименования.

Такие системы отслеживания мнений авторов по их публикациям и текстам могут быть использованы для решения множества практических задач. Например, системы **Opinion Mining** могут:

- 1) являться дополнительным источником предсказания изменений на фондовом рынке или возможных изменений стоимости акций;
- 2) помочь корпорациям определить пользовательское мнение о новом продукте, например о новом iPhone;
- 3) помочь политикам определить предпочтения избирателей на предстоящих выборах.

К этой же группе задач относится другое важное приложение лингвистического процессора – **машинный перевод**, подразумевающий **полностью автоматический переводчик**. В автоматически получаемых переводах, несмотря на достигнутый в последнее время прогресс в развитии технологии, до сих пор встречается достаточное количество ошибок, которые подлежат ручному редактированию.

Последний (третий) класс задач NLP – это задачи, которые на сегодня находятся только на уровне научных исследований и экспериментальных разработок. К таким задачам можно отнести разработку вопросно-ответных систем, которые используются для автоматического ответа на вопросы любого вида. Question answering systems имеют реальные алгоритмы только при формулировке самых простых, фактографических вопросов. Даже простые, но общие вопросы по-прежнему тяжело поддаются автоматизации.

Нерешенной задачей NLP остается задача автоматической генерации онтологии (**Ontology generation**, или ontology acquisition), включающая автоматическое извлечение из текстов терминов данной предметной области и автоматическое

выявление отношений между терминами на базе корпуса текстов.

Еще одной, практически нерешенной задачей лингвистического процессора, является **перефразирование (paraphrases)**. Например, такая система должна правильно принять решение о том, что предложение «*тринадцать солдат потеряли свои жизни*» имеет тот же смысл, что и предложение «*часть войска была разгромлена*». Такая система перефразирования должна обладать мощными средствами семантического смыслового анализа, практически не существующими на сегодняшний день.

Система **автоматического реферирования (summarization)**, которая к сложностям перефразирования добавляет проблемы смыслового обобщения, также не имеет практической реализации. В настоящее время на рынке по-прежнему присутствуют только системы квазиреферирования, использующие по большей части статистико-позиционные подходы, а алгоритмы «истинного» смыслового обобщения практически отсутствуют. Такая обобщающая система, получая на вход текстовую информацию о существующих в Португалии финансовых проблемах, греческих долговых обязательствах и итальянском долге, на выходе должна сделать вывод, что все это подобно европейскому долговому кризису.

Существует достаточное количество проблем и особенностей естественного языка, не позволяющих быстро и легко автоматизировать его обработку. К таким проблемам, прежде всего, необходимо отнести многозначность естественного языка. Под многозначностью понимают любой случай, когда внешнее выражение элемента языка (знака) может иметь множественную интерпретацию. Неопределенность или многозначность, при обработке текстов, можно выделить на графемном, морфологическом, синтаксическом и семантическом уровнях языковой системы. Еще одной особенностью языка является его идиоматичность. Под идиомой понимается фраза, значения которой отличается от значения, которое можно ожидать из суммы значений слов данной фразы.

Кроме того, существуют проблемы, связанные с сегментацией (выделением) слов и предложений, проблемы выделения имен сущностей, синонимов, ко-референтности. С появлением социальных сетей и повышением скорости распро-

странения текстовой информации к ранее существующим проблемам наличия большого количества ошибок и опечаток в текстах добавилась проблема использования нестандартного языка, который часто можно видеть в текстах подобных Twitter, SMS, blog, социальных сетях и т. д. Например, когда используются большие буквы или когда слова пишутся с использованием символов 2, U, ♥ и им подобных.

Можно выделить следующие технологии компьютерной лингвистики, которые направлены на решение этих и других проблем «понимания» языка:

- Part-of-speech (POS) tagging (автоматическая разметка частей речи, морфологическая обработка, морфологический анализ);
- Parsing (парсинг, синтаксическая обработка);
- Name entity recognition (распознавание сущностей (концептов));
- Relation Extraction (распознавание отношений между сущностями);
- Co-reference resolution (решение задачи снятия кореферентности, распознавание анафор и катафор);
- Word sense disambiguation (снятие смысловой многозначности слов);
- Collocation extraction (извлечение словосочетаний);
- Semantic Equivalence recognition (распознавание семантических эквивалентов).

От того, насколько успешно работают алгоритмы данных этапов обработки текста, зависит успешность работы тех или иных приложений NLP.

1.2. Установка открытой библиотеки NLTK

Открытая библиотека Natural Language Toolkit (NLTK), которую можно вместе с данными и документами загрузить с сайта <http://www.nltk.org/>, представляет собой среду, используемую для построения NLP программ на Питоне. Она включает базовые классы для представления данных, связанных с NLP, и стандартный интерфейс для выполнения таких задач, как POS-tagging, syntactic parsing, text classification и др.

Библиотека NLTK была разработана в 2001 году как часть курса Computa-

tional Linguistics course на кафедре Computer and Information Science университета Pennsylvania. Все последующее время он дополнялся и использовался во многих университетах и исследовательских проектах. Таблица 1.1 показывает наиболее часто используемые NLTK-модули.

Таблица 1.1 – Основные задачи NLP и соответствующие им модули пакета NLTK

Решаемая задача NLP	NLTK-модуль	Функции модуля
Доступ к корпусам	nltk.corpus	Стандартизированный интерфейс корпусов и лексиконов
Обработка строк	nltk.tokenize, nltk.stem	Токенизация, сегментация предложений, стемменг
Работа с коллокациями	nltk.collocations	Использование метрик t-test, chi-squared, PMI
Part-of-speech разметка	nltk.tag	n-граммы, backoff, скрытая марковская модель, TnT
Классификация	nltk.classify, nltk.cluster	Дерево решений, максимальная энтропия, наивный Байес, k-средние, максимальное правдоподобие
Синтаксический разбор предложений	nltk.chunk	Функциональные и вероятностные зависимости
Метрики оценки	nltk.metrics	Полнота, точность, коэффициент согласованности
Вероятность и оценка	nltk.probability	Частотные распределения, сглаженные вероятностные распределения
Приложения	nltk.app, nltk.chat	Графический конкодансер, парсер, WordNet-браузер, chatbots

Для установки инструментария NLTK необходимо зайти на официальную страницу <http://www.nltk.org/>, выбрать пункт меню **Installing NLTK** и скачать на компьютер необходимую инсталляцию пакета.

Для того чтобы проверить версию Python, установленного на компьютере, можно в консольном режиме использовать команду `python –version`.

После того как пакет будет скачан на компьютер, можно установить его с

помощью установщика pip: `pip install nltk`.

После установки можно ввести две команды, позволяющие установить на компьютере “book collection” *NLTK-Data*, представляющие лингвистические корпуса для обработки:

```
>>>import nltk
```

```
>>>nltk.download()
```

При загрузке *NLTK Book Collection* на экран будет выведено окно, отражающее доступные для использования пакеты. Вкладка *Collections NLTK Downloader* показывает, как пакеты сгруппированы во множества (рис. 1.1).

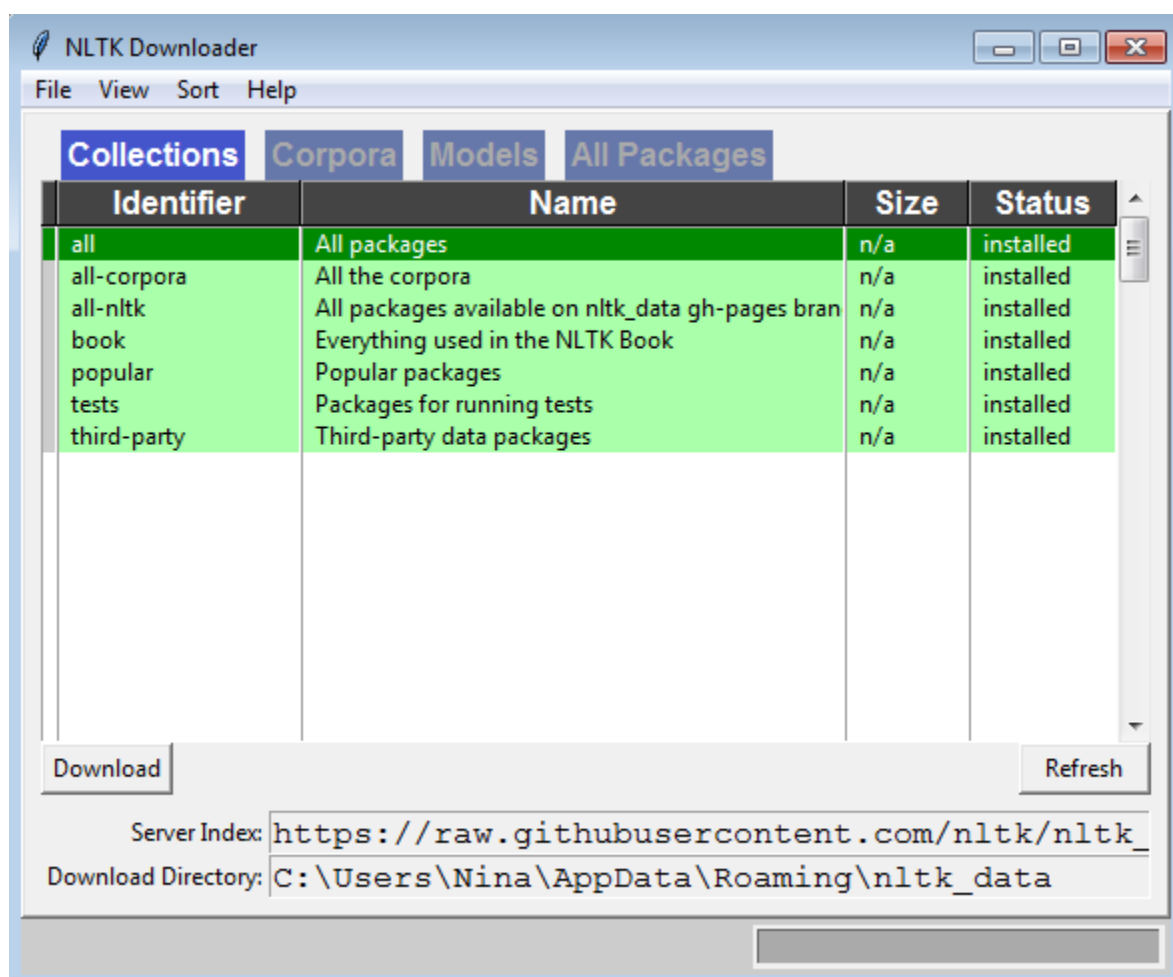


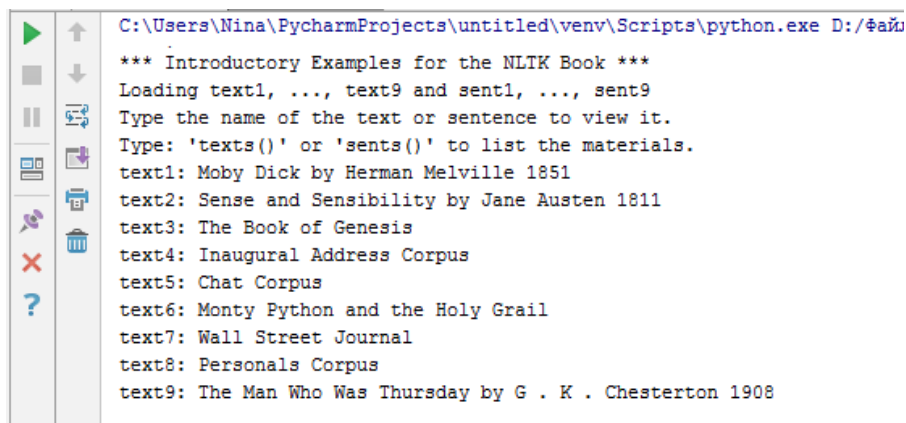
Рисунок 1.1 – Пакеты корпусов, доступные при установке NLTK

Все пакеты требуют примерно 2.8 Гбайт дискового пространства.

После того как данные корпусов будут загружены на компьютер, можно их

обрабатывать с помощью интерпретатора Python. Для того чтобы увидеть названия книг, тексты которых предлагает *NLTK-Data*, используется команда `from nltk.book import *`.

После этого на экран выводится список книг, тексты которых доступны для обработки:

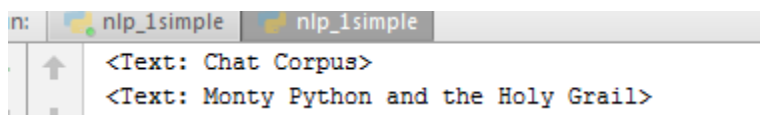


```
C:\Users\Nina\PycharmProjects\untitled\venv\Scripts\python.exe D:/#ай
*** Introductory Examples for the NLTK Book ***
Loading text1, ..., text9 and sent1, ..., sent9
Type the name of the text or sentence to view it.
Type: 'texts()' or 'sents()' to list the materials.
text1: Moby Dick by Herman Melville 1851
text2: Sense and Sensibility by Jane Austen 1811
text3: The Book of Genesis
text4: Inaugural Address Corpus
text5: Chat Corpus
text6: Monty Python and the Holy Grail
text7: Wall Street Journal
text8: Personals Corpus
text9: The Man Who Was Thursday by G . K . Chesterton 1908
```

Если необходимо получить информацию о каком-то конкретном тексте, достаточно написать его имя:

```
print(text5)
```

```
print(text6)
```



```
n: nlp_1simple nlp_1simple
<Text: Chat Corpus>
<Text: Monty Python and the Holy Grail>
```

Контрольные вопросы и практические задания к теме “Введение в Natural Language Processing”

1. Существующие приложения NLP.
2. Современные проблемы автоматизации обработки текстов.
3. Наиболее применяемые модули пакета NLTK.
4. Что понимается под понятием Natural Language Processing?
5. Приложения NLP, имеющие коммерческую реализацию.
6. Приложения NLP, существующие только в экспериментальных разработках.
7. Что собой представляет пакет NLTK Python?

Тема 2. СТРОКИ PYTHON

2.1. Особенности работы с текстовыми строками в Python.

2.2. Форматирование строк Python.

2.3. Строки в формате Unicode в Python.

2.4. Строковые методы и функции.

Контрольные вопросы и практические задания по теме «Строки Python».

2.1. Особенности работы с текстовыми строками в Python

Строки представляют собой пример последовательности в Python — это последовательность символов с произвольным доступом. Для работы со строками Питон имеет встроенный строковый класс, называемый **str**. Можно управлять текстовыми данными в Питоне, используя строковые объекты. Строка представляет собой неизменяемую последовательность *Unicode*. Поэтому измененную строку можно скопировать или сохранить в другой строке.

Записываются строки в двойных или одинарных кавычках. В случае необходимости в программном коде длинные строки можно разбивать с помощью обратного слеша или вводить их с помощью тройных кавычек или тройных апострофов:

```
>>> s = 'this is first word\
and this is second word'
>>> c=""" Многострочный блок
текста """
```

При работе со строками обратный слеш \ также может быть использован для обозначения управляющих символов (\n, \t, \f – перевод строки, \xhh – шестнадцатеричное значение, \ooo – восьмеричное значение).

Оператор + со строками определяет конкатенацию строк. Кроме того, можно объединять строки, просто расположив их рядом:

```
>>> print ("Hello " "World")
```

Оператор * (умножить) может использоваться для повторения строк:

```
>>> print ("Hello " * 3)
```

Так как строка – это последовательность символов, с ней можно работать, используя срезы и индексы, подобно списками и кортежам.

Можно получить любой символ строки по его индексу. Индекс крайнего слева символа равен нулю, следующего – единице и т. д. Можно получить также доступ к символам строки, используя индексы противоположного направления, отсчитываемые от конца строки (от минус один) (рис.2.1).

инд	0	1	2	3	4	5	6	7	8	9
str="	0	1	2	3	4	5	6	7	8	9"
инд	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

Рисунок 2.1 – Возможные индексы символов строки.

Так как каждый символ строки Питона имеет соответствующий индекс, можно манипулировать строками так же, как и другими типами последовательностей:

```
>>>print ("Hello " [0])
'Н'
>>>x='Hello'
>>>print (x [-1])
'o'
```

Если указать смещение, равное длине строки и больше, то сгенерируется исключение.

Срезы представляют собой механизм Питона, позволяющий получить элемент или подмножество элементов из контейнера объектов, используя значения индексов. Таким образом, срезы позволяют избежать использования циклов для того, чтобы получить доступ к некоторым подстрокам.

Срез представляет собой оператор **[start:end:step]**, позволяющий извлечь подстроку из строки. В подстроку будут включены символы с номера **start** до **end-1**, опуская символы, чье смещение внутри подстроки кратно (т. е. с шагом)

step:

- `[:]` извлекает всю последовательность;
- `[start:]` извлекает последовательность с символа номер **start** до конца строки;
- `[:end]` извлекает последовательность от начала строки до **end-1**;
- если задать отрицательное смещение, Python будет двигаться в обратную сторону:

```
>>>letter='abcd'
>>>print(letter[2:3])
'c'
>>>print (letter[:2])
'ab'
>>>print (letter[:2])
'ac'
>>>s=r'\n\n\'
>>>print (s[:-1])
\n\n\
```

Если задать отрицательный шаг, то можно получить реверсное представление строки. При шаге минус 1, строка формируется от конца и затем выводит один символ за другим:

```
number_str='123456789'
print(number_str[::-1])
print(number_str[::-2])
```

987654321
97531

2.2. Форматирование строк Python

При выводе информации на экран Питон позволяет интерполировать данные в строку, т. е. разместить значения внутри строк, используя различные форматы. Питон поддерживает два основных способа **форматирования строки**, которые часто называют *старым стилем* и *новым стилем* форматирования. Оба стиля поддерживаются как Python 2, так и Python 3.

Старый стиль форматирования проще, в нем строка имеет вид

строка % данные

Встроенная операция % называется оператором интерполяции (*interpolation operator*) и может работать как с одной, так и со множеством переменных:

‘ строка_символов %f₁ строка символов’ % переменная1

или

‘ строка_символов %f₁ %f₂строка символов’ %(переменная1, переменная2...),

где *f* определяет формат и может представлять собой:

'%d', '%i', '%u'	десятичное число
'%o'	число в восьмеричной системе счисления
'%x'	число в шестнадцатеричной системе счисления (буквы в нижнем регистре)
'%X'	число в шестнадцатеричной системе счисления (буквы в верхнем регистре)
'%e'	число с плавающей точкой с экспонентой (экспонента в нижнем регистре)
'%E'	число с плавающей точкой с экспонентой (экспонента в верхнем регистре)
'%f', '%F'	число с плавающей точкой (обычный формат)
'%c'	символ (строка из одного символа или число - код символа)
'%s'	строку

Например,

```
print ('%d %s и %5.1f %s' % (6, 'бананов', 12.5, 'lemons'))  
print ('%d %s и %1.1f %s' % (6, 'бананов', 12.5, 'lemons'))
```

```
6 бананов и 12.5 lemons  
6 бананов и 12.5 lemons
```

Новый стиль форматирования создается с помощью символов `{}` и функции **format**. В Python 3 рекомендуется применять новый стиль форматирования, не использующий оператор `%`.

Форматирование осуществляется вызовом функции **.format()** для объекта строки:

```
text='{ } and { } = { }'.format('teacher','student', 345)  
print (text)
```

Аргументы старого стиля нужно ставить в порядке их появления в качестве заполнителей символа `%`. В новом стиле можно указывать порядок значений в строке при помощи индексов:

```
text='{2} = {1} and {0}'.format('teacher','student', 345)  
print (text)
```

```
345 = student and teacher
```

Аргументы функции **format ()** могут быть словарем или именованными аргументами, а спецификаторы могут включать их имена:

```
print('{num} = {teach} and {stud}'.format(teach='Professor', \  
stud='Ivanov', num=100))
```

```
100 = Professor and Ivanov
```

В приведенных выше примерах аргументы выводятся на экран с форматированием по умолчанию. Старый стиль позволяет указать спецификатор формата после символа `%`, а новый стиль – после двоеточия, после которого можно установить минимальную длину поля, максимальную ширину символов:

```
print('{2:10d} = {1:10f} and {0:10.1f}'.format(23,36.34, 47))
print('{2:d} = {1:10f} and {0:10.1f}'.format(23,36.34, 47))
```

```
          47 =  36.340000 and          23.0
47 =  36.340000 and          23.0
```

Для типов *float* и *string* можно установить количество цифр, выводимых после запятой (для *string* указывается максимальное количество выводимых символов). Для целых чисел дробную часть в формате указывать нельзя: **{0:10.1d}**

```
print("{0:>10.4s} = {1:>10.3s}".format("123456789", "0987654321"))
```

```
      1234 =      098
```

Использование символов *>*, *<* и *^* позволяет указать выравнивание в поле вывода по правому краю, по левому краю и по центру:

```
print('{2:<10d} = {1:>10.1f} and {0:>10.1f}'.format(23,36.34, 47))
print('{2:>10d} = {1:>10.1f} and {0:>10.1f}'.format(23,36.34, 47))
print('{2:^10d} = {1:^10.1f} and {0:^10.1f} и {3:^10}'.format(23,36.34, 47, 'текст'))
```

```
47 =  36.340000 and          23.0
47          =          36.3 and          23.0
    47          =   36.3    and    23.0    и    текст
```

Вместо пробела при выводе можно использовать символ-заполнитель, который надо разместить сразу после двоеточия, но перед символами выравнивания (*>*, *<*, *^*):

```
print ('{0:!!^20s}'.format('BIG SALE'))
```

```
!!!!!!BIG SALE!!!!!!
```

2.3. Строки в формате Unicode в Python

Строки в Python 3 являются строками формата Unicode, а не массивом байт. Формат Unicode разработан для возможности представления всех символов языков. В юникоде каждая буква или символ представляется 4-байтовым числом.

Unicode — это действующий международный стандарт, который предоставляет уникальный номер каждому символу всех языков мира, плюс математические и другие символы, независимо от платформы, программы и языка.

В предыдущей версии Питона и во многих языках программирования строки имеют формат ASCII (*American standard code for information interchange*), при котором каждый символ хранится в одном байте, 8 битах. Таким образом, всего 256 возможных вариантов.

2.4. Строковые методы и функции

Существуют специальные строковые методы, которые вызываются как **строка.метод(аргументы)**.

Так как строки представляют собой неизменяемый объект, то такие методы возвращают новую строку, которую следует присвоить переменной.

Метод **split()** позволяет разбить строку на список строк, основываясь на разделителе:

```
>>>stroka="Give, me, please, a, pen"
>>>a=stroka.split(',')
>>> print(a)
```

['Give', 'me', ' please', ' a', ' pen']

Если не указать параметр, т. е. не указать разделитель, метод **split()** будет для разделения использовать любую последовательность пробелов, в том числе символы перехода на новую строку и табуляции.

Метод **join()** является методом, противоположным методу **split()**. Он предназначен для объединения элементов списка строк в одну строку. Этот метод является методом символа разделителя, а параметром метода является список подстрок, которые объединяются с помощью разделителя в строку:

```
string_glues.join(list)
print (' '.join(['Could','you', 'please']))
```

Метод **startswith(prefix[, start[, end]]) -> bool** определяет, начинается ли строка с префикса **prefix** (который может быть задан кортежем). Поиск начинается с символа с индексом **start** и заканчивается символом с индексом **end**:

```
poem='У лукоморья дуб зеленый\  
  \n Златая цепь на дубе том'
```

```
print (poem) True
```

```
print(poem.startswith('У'))
```

```
print(poem.startswith('ком',4,8))
```

Метод **endswith(suffix[, start[, end]])** -> **bool** определяет, заканчивается ли строка указанным постфиксом; **suffix** так же, как **prefix**, может быть указан кортежем:

```
print(poem.endswith(('том','там','ья'), 5,11)) True
```

Метод **find(sub[, start[, end]])** -> **int** – находит подстроку в строке и возвращает либо позицию первого вхождения подстроки, либо -1, если подстрока не найдена; **start=0** – индекс начала среза в исходной строке, с которой начинается поиск подстроки.

Метод **rfind(sub[, start[, end]])** -> **int** – находит подстроку в строке – возвращает позицию последнего вхождения строки.

```
stroka='word word word'
```

```
print (stroka.find('word'))
```

```
print (stroka.rfind('word'))
```

0
10

Метод **count(sub[, start[, end]])** -> **int** – возвращает количество вхождений подстроки в строку:

```
print(stroka.count('wor'))
```

```
print(stroka.count('wor',4,10))
```

3

1

Метод **replace(old, new[, maxcount])** -> **str** – возвращает копию строки, в которой подстрока для замены **old** заменяется на новую подстроку **new**, **maxcount** показывает максимальное количество замен. Если этот параметр не указан, то осуществляются все возможные замены:

```
stroka='word word word'
```

```
print (stroka.replace('d','ld',2))
```

world world word

Метод **isalnum()** -> **bool** – возвращает True, если в строке есть хотя бы один символ и все символы строки являются цифрами или буквами, иначе возвращает False:

```
print('word word'.isalnum())
```

False

```
print('WWW5678'.isalnum())
```

True

Метод **isalpha()** -> **bool** – возвращает True, если строка содержит только буквы, иначе – False:

```
print('WWW5678'.isalpha())
```

False

```
print('ДубЗеленый'.isalpha())
```

True

Метод **isspace()** -> **bool** – возвращает True, если в строке есть только пробельные символы (пробел, табуляция, пустая строка и др.)

Метод **isdigit()** -> **bool** – возвращает True, если строка состоит только из цифр.

Метод **strip([chars])** -> **str** – возвращает копию строки, с обоих концов которой устранены указанные символы. Если параметр не указан, то удаляются пробельные символы. Перечисленные в качестве параметра символы представляют список отдельных символов, а не последовательность символов:

```
setup=' a duck goes into a bar....\n'
```

```
print (setup, setup.strip(' \n '))
```

```
print ( 'abca'.strip('ac') ) # 'b'
```

```
a duck goes into a bar....|
a duck goes into a bar
```

Два аналогичных метода **lstrip([chars])** -> **str** и **rstrip([chars])** -> **str** – возвращают копию указанной строки, у которой удалены заданные символы с начала (слева l – left) и с конца строки соответственно.

Метод **capitalize()** – возвращает копию строки, переводя первую букву в верхний регистр, а остальные в нижний.

Метод **title()** – возвращает копию строки; первая буква каждого нового слова становится заглавной, в то время как остальные становятся строчными. Такое написание характерно для заголовков англоязычных статей.

Метод **upper()** – возвращает копию исходной строки с символами, приведёнными к верхнему регистру.

Метод **islower()** – возвращает True, если все символы строки прописные.

Метод **lower()** – возвращает копию строки с символами, приведёнными к нижнему регистру.

Метод **swapcase()** – возвращает копию строки, в которой каждая буква будет иметь противоположный регистр.

Длину строки можно получить с помощью встроенной функции **len()**, которая используется для строки и других типов последовательностей.

Контрольные вопросы и практические задания по теме «Строки Python»

1. Особенности работы с текстовыми строками в Питоне.
2. Использование экранированных последовательностей и «сырых» строк в Питоне.
3. Использование механизма срезов при работе со строками.
4. Форматирование строк с помощью оператора интерполяции.
5. Встроенные методы работы со строками.
6. Программно создайте текстовый файл `file1.txt`, в который запишите фрагмент стихотворения, в заданном формате:

O, speak again, bright angel, for thou art
As glorious to this night, being o'er my head,
As is a winged messenger of heaven
Unto the white-upturned wondering eyes
Of mortals that fall back to gaze on him
When he bestrides the lazy-puffing clouds
And sails upon the bosom of the air.

Перепишите в другой файл `file2.txt` все строки, которые начинаются на “А” и “О”. Создайте для строковых преобразований функцию. Подсчитайте количество записанных строк. Выведите на экран содержимое первого или второго файла, в зависимости от имени файла введенного пользователем. В случае неверного ввода имени выведите соответствующее сообщение. Вывод осуществлять без пустых строк.

7. Программно создайте текстовый файл `file1.txt`, в который запишите фрагмент стихотворения задания 1. Перепишите в другой файл `file3.txt` данный

фрагмент, заменив все строки, которые начинаются с “As”, на пустые строки. Создайте для строковых преобразований функцию. Подсчитайте количество записанных строк. Выведите на экран содержимое первого или второго файла, в зависимости от имени файла введенного пользователем. В случае неверного ввода имени выведите соответствующее сообщение. Вывод осуществлять без пустых строк.

8. Программно создайте текстовый файл file1.txt, в который запишите фрагмент поэмы в заданном формате:

What's Montague? It is nor hand, nor foot,
Nor arm, nor face, nor any other part
Belonging to a man. O, be some other name.
What's in a name? That which we call a rose
By any other name would smell as sweet;

Посчитайте, сколько раз слово *name* встретилось в данном фрагменте. Перепишите в другой файл file2.txt этот же текст, заменив слово name на написанное с большой буквы слово Surname. Создайте для строковых преобразований функцию. Выведите на экран содержимое первого или второго файла, в зависимости от имени файла введенного пользователем. В случае неверного ввода имени выведите соответствующее сообщение. Вывод осуществлять без пустых строк.

9. Программно создайте текстовый файл file1.txt, в который запишите фрагмент поэмы в заданном формате:

Thou blind fool, Love, what dost thou to mine eyes
That they behold, and see not what they see
They know what beauty is, see where it lies
Yet what the best is take the worst to be

Создайте функцию change_str(), которая, если строка параметра заканчивается на согласную, добавляет данную согласную и букву “ay” в конце строки, а если строка заканчивается на гласную, то в конце строки добавляется слово “way”. Перепишите в другой файл file2.txt измененный функцией текст файла file1.txt. Выведите на экран содержимое первого или второго файла, в зависимости от имени

файла введенного пользователем. В случае неверного ввода имени выведите соответствующее сообщение. Вывод осуществлять без пустых строк.

10. Программно создайте текстовый файл file1.txt, в который запишите следующую строку об отправителе полученного электронного сообщения:

From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2018

Извлеките из текста данного файла file1.txt и запишите в файл file2.txt имя хоста отправителя сообщения, зная, что хост записывается после символа @ до ближайшего пробела. Выведите на экран содержимое первого или второго файла, в зависимости от имени файла введенного пользователем. В случае неверного ввода имени выведите соответствующее сообщение. Вывод осуществлять без пустых строк.

11. Программно создайте текстовый файл file1.txt, в который запишите следующий текст электронного сообщения:

From stephenev.martin@net.ac.za Sat Jan 5 10:15:46 2020

X-Authentication-Warning: set sender to stephenev.martin@net.ac.za using – f

From stephenev.martin@net.ac.za

Author: stephenev.martin@net.ac.za

From anton.horwichev@gmail.com Fri Jan 4 08:25:23 2020

X-Authentication-Warning: set sender to santon.horwichev@gmail.com using – f

Перепишите в другой файл file2.txt все строки, которые не содержат наименования хоста @uct.ac.za. Выведите на экран содержимое первого или второго файла, в зависимости от имени файла введенного пользователем. В случае неверного ввода имени выведите соответствующее сообщение. Вывод осуществлять без пустых строк.

12. Программно создайте текстовый файл file1.txt, в который запишите следующий форматированный текст:

Бананы 12.23 Ананасы 3.40

Кокосы 124.23 Яблоки 233.40

Лимоны 789.23 Гранат 55.71

Виноград 5.20 Сливы 233.40

Перепишите в другой файл file2.txt все строки, которые заканчиваются нулем. Посчитайте эти строки. Выведите на экран содержимое первого или второго файла, в зависимости от имени файла введенного пользователем. В случае неверного ввода имени выведите соответствующее сообщение. Вывод осуществлять без пустых строк.

13. Программно создайте текстовый файл file1.txt, в который запишите следующий:

```
d:/ФайлыPython/files/file1.txt Log:Admin1 Password:12345
```

```
send message saturday january 2018 Horwichev
```

Перепишите в другой файл file2.txt все слова, не содержащие специальные знаки, а содержащие только буквы и цифры. Все слова записывать с большой буквы. Отдельной строкой добавьте в файл file2.txt пароль, зная, что он записан после слова Password: и до ближайшего пробела. Выведите на экран содержимое первого или второго файла, в зависимости от имени файла введенного пользователем. В случае неверного ввода имени выведите соответствующее сообщение. Вывод осуществлять без пустых строк.

Тема 3. РЕГУЛЯРНЫЕ ВЫРАЖЕНИЯ

3.1. Введение в регулярные выражения.

3.2. Использование регулярных выражений в Python.

3.3. “Нежадные” квантификаторы в Python.

3.4. Круглые скобки в регулярных выражениях Python.

3.5. Методы и функции модуля re.

Контрольные вопросы и практические задания по теме «Регулярные выражения».

3.1. Введение в регулярные выражения

Наиболее фундаментальным, хорошо разработанным инструментом автоматической обработки текстов (*Text processing*) являются регулярные выражения (*regular expression*). Регулярные выражения в общем смысле – это система обработки текста, основанная на специальной системе записи образцов для поиска.

Регулярное выражение в практическом, узкоспециальном подходе, — это строка, имеющая специальный синтаксис, которая представляет шаблон какой-либо искомой строки. Такой шаблон является последовательностью букв или символов, которые могут отобразить или текст, или слово, или пунктуацию. Регулярные выражения позволяют осуществить сложный поиск или замену в строке или ее разбиение на части различными способами.

Например, регулярные выражения позволяют найти подстроку, подобную URL веб-странице, e-mail-адресу, или удалить нежелательные строки или символы.

Существует несколько стандартов регулярных выражений, правила грамматики которых могут немного отличаться:

- **ECMAScript** (подобен JavaScript) – представляет собой встраиваемый скриптовый язык, не имеющий средств ввода-вывода, и используется для построения других скриптовых языков (JavaScript – имплементация стандарта ECMAScript компанией Mozilla, JScript – имплементация стандарта ECMAScript компанией Wicrosoft).

- POSIX-совместимые языки регулярных выражений (*Portable Operating System Interface for Unix*) – представляют собой переносимый интерфейс операционных систем Unix). Считаются устаревшими и медленными.

- PCRE ((Perl Compatible Regular Expressions) – библиотека, реализующая работу регулярных выражений в стиле Perl).

Регулярное выражение представляет собой образец (*pattern*), задающий правило поиска, который также называют «шаблоном» или «маской». Каждый символ в шаблоне проверяется на соответствие символу в целевой последовательности, т. е. в последовательности один за другим. В шаблон могут быть включены как обычные символы (оригинальные символы, которые также будут в *target sequence*), так и метасимволы – символы, которые имеют специальные значения, определяющие символы или классы символов в целевой последовательности.

Простейший инструмент в регулярных выражениях – дизъюнкция. Например, метасимвол “[]” (квадратные скобки) применяется для описания подмножеств, и внутри регулярного выражения рассматривается как один символ, который может принимать значения, перечисленные внутри этих скобок.

Предположим, что нужно найти в тексте фамилию “Иванов”, но существует различные возможные способы записи: *Иванову*, *Ивановы* и т. д. Так как квадратные скобки в шаблоне регулярного выражения обозначают любой символ внутри этих строк, то шаблон будет иметь следующий вид: *Иванов[ауы]*

Различают два множества *метасимволов*: те, что распознаются в любом месте шаблона, за исключением содержимого квадратных скобок, и те, что распознаются внутри квадратных скобок.

В квадратных скобках можно указать символы, которые могут встречаться на этом месте строки. Можно перечислять символы подряд или указывать диапазон через тире:

[09]	соответствует числу 0 или 9
[0-9]	соответствует любому числу от 0 до 9
[абв]	соответствует буквам “а”, “б” и “в”
[а-г]	соответствует буквам “а”, “б”, “в” и “г”

[а-яё]	соответствует любой букве от “а” до “я”
[а-яА-ЯёЁ]	соответствует любой букве от “а” до “я” в любом регистре

Отрицание в дизъюнкции используется в том случае, когда шаблон не должен содержать символы. Если первым символом внутри скобок является ^ (*caret*), то значением символьного класса могут быть только символы, НЕ перечисленные внутри скобок:

[^09]	не цифра 0 или 9
[0-9]	не цифра от 0 до 9
[а-яА-ЯёЁ]	не буква

Символ “^” теряет свое специальное значение, если он не расположен сразу после открывающейся квадратной скобки. Кроме того, для того чтобы отменить специальное значение символа “–” (дефиса), его необходимо указывать после перечисления всех символов, перед закрывающейся квадратной скобкой. Точка также теряет свое специальное значение в квадратных скобках и обозначает только символ точки.

К метасимволам регулярного выражения, распознаваемым вне квадратных скобок, относятся такие специальные символы, как ., ^, \$, *, +, ?, {, }, [,], \, |, (,).

метасимвол	значение
\	меняет тип символа, следующего за ним, на противоположный, т. е. если это был обычный символ, то он МОЖЕТ превратиться в метасимвол, а если это был метасимвол, то он теряет свое специальное значение и становится обычным символом. Например, символ "." в обычном режиме означает "любой единственный символ", а символ "\" – просто точку.

метасимвол	значение
\	обозначает генерируемый символьный тип (т. е. содержит обозначение класса возможных значений). Сюда входит:
\d	– любая десятичная цифра (0–9);
\D	– любой символ, кроме десятичной цифры;
\s	– любой пустой символ (пробел или табуляция);
\S	– любой символ, кроме пустого;
\w	– символ, используемый для написания Perl-слов (это буквы, цифры и символ подчеркивания), так называемый "словарный символ". "[a-zA-Z0-9_]"*
\W	– несловарный символ (все символы, кроме определяемых \w). Обозначение аналогично записи "[^a-zA-Z0-9_]"*
\	обозначает кодирование непечатных символов (аналогичных escape-последовательностям языка C) :
\\	– бек-слеш (backslash);
\n	– символ перевода строки;
\e	– символ escape;
\t	– символ табуляции.
\xhh	обозначает шестнадцатеричный код (Hexadecimal), где hh – две шестнадцатеричных цифры (0–9 a–f A–F). Например, “\x41” обозначает букву A.
^	обозначается в начале строки. Объявляет начало объекта (или строки в многострочном режиме). Например, шаблон “^abc” обозначает, что строка начинается с “abc”.
\$	Маркер конца строки (или текста в многострочном режиме). Например, шаблон “abc\$” обозначает, что строка заканчивается на «abc».
()	Круглые скобки применяются для выделения групп или подшаблонов.

Если такие специальные символы необходимо трактовать “как есть” (т. е. как обычные символы), то их следует экранировать с помощью слеша. Некоторые специальные символы теряют свое специальное значение, если их разместить внутри квадратных скобок, тогда их не нужно экранировать.

Например, символ «точка» соответствует любому символу, кроме символа перевода строки. Если необходимо найти именно точку, то перед ней надо указать символ `\` или поместить точку внутри квадратных скобок.

К метасимволам, записываемым вне квадратных скобок, относятся также квантификаторы. Квантификаторы — метасимволы, описывающие количественные или повторяющиеся отношения. Существует два типа квантификаторов:

- общие, которые задаются с помощью фигурных скобок;
- сокращенные, представляющие собой исторически сложившиеся сокращения наиболее распространенных квантификаторов.

Общие квантификаторы:

`{n}` — повторение предыдущего символа точно n раз.

`{n,}` — повторение предыдущего символа n и более раз.

`{min,max}` — повторение предыдущего символа от min до max раз.

Сокращенные квантификаторы:

`*` — ноль или большее число вхождений символа в строку; эквивалентно комбинации `{0,}`;

`+` — одно или большее число вхождений символа в строку; эквивалентно комбинации `{1,}`;

`?` — ни одного или одно вхождение символа в строку; эквивалентно комбинации `{0,1}`.

Все квантификаторы являются «жадными» — при поиске соответствия шаблону ищется самая длинная подстрока, соответствующая шаблону, и не учитываются более короткие соответствия.

3.2. Использование регулярных выражений в Python

По сути регулярные выражения — это язык программирования, встроенный в Python и доступный при помощи модуля `re`. Используя его, можно указать прави-

ла для множества возможных строк, которые необходимо найти или проверить

Функция `compile()` модуля **re** позволяет создать откомпилированный шаблон регулярного выражения и имеет синтаксис

re.compile (регулярное_выражение [, модификатор])

В данном выражении модификатором может быть:

- **I** или **IGNORECASE** – означает поиск без учета регистра.
- **L** или **LOCALE** – учитывает настройки текущей локали. Для русского и украинского языков необходимо указать данный флаг и настроить локаль.
- **S** или **DOTALL** – означает, что метасимвол «точка» будет соответствовать любому символу, включая символ перевода строки (`\n`). По умолчанию символ «точка» не соответствует символу перевода строки.
- **M** или **MULTILINE** – означает поиск в строке, состоящей из нескольких подстрок, разделенных символом новой строки (`\n`). Символ `^` соответствует привязке к началу каждой подстроки, а символ `$` - позиции перед символом перевода строки.
- **X** или **VERBOSE** – означает, что если флаг указан, то пробелы и символы перевода строки будут игнорированы. Кроме того, внутри регулярного выражения можно использовать комментарий.
- **U** или **UNICODE** – означает, что шаблон и строка ввода рассматриваются в кодировке Unicode.

Традиционно перед всеми строками, содержащими регулярные выражения, указывается модификатор **r**, показывающий, что используются неформатированные строки (*raw string*). В таком случае заданный шаблон будет интерпретироваться как строка. Иначе строка, использующая бек-слеш может интерпретироваться как *escape sequences*, т. е. “`\n`” будет являться индикатором перехода на новую строку.

Если модификатор не указан, то все слешы необходимо будет экранировать. Например, строку

```
p=re.compile(r “^\w+$”)
```

нужно будет записать как

```
p=re.compile("^\\w+$")
```

3.3. «Нежадные» квантификаторы в Python

Так как все квантификаторы являются «жадными», то если для поиска содержания тега `` будет использован следующий, наиболее очевидный, шаблон "` Text1Text2Text3`", то в результате будет “захвачен” значительно больший фрагмент текста, чем необходимо.

```
s=" <b> Text1</b>Text2<b>Text3</b>"
```

```
p=re.compile(r"<b>.*</b>",re.S)
```

```
print(p.findall(s))
```

В результате будет возвращена подстрока: `[' Text1Text2Text3']`

Чтобы ограничить «жадность» квантификаторов, необходимо после квантификатора указывать символ вопроса “?”:

```
s=" <b> Text1</b>Text2<b>Text3</b>"
```

```
p=re.compile(r"<b>.*?</b>",re.S)
```

```
print(p.findall(s))
```

В этом случае квантификатор “*” не будет “захватывать” всю строку и в результате будет возвращена искомая подстрока `[' Text1', 'Text3']`

3.4. Круглые скобки в регулярных выражениях Python

Круглые скобки используются для группировки фрагментов внутри шаблона и часто со специальным символом `|` -или. Например, фрагмент программного кода:

```
str="abcabc 23 privet!"
```

```
p=re.compile(r"abcabc \d{2}")
```

```
print(p.findall(str))
```

```
['abcabc 23']
```

можно записать, выделив круглыми скобками дважды повторяемую часть регулярного выражения

```
p=re.compile(r"((abc){2} \d{2})")
```

Но круглые скобки «вырывают» (возвращают) куски выделенных строк. То есть в результате такого шаблона будет получен следующий кортеж совпада-

ющих подстрок:

```
[('abcabc 23', 'abc')]
```

Если в регулярном выражении необходимо сгруппировать какую-то часть, но не возвращать ее, то после открывающейся скобки, подстроку которой не нужно возвращать, необходимо поставить знак вопроса и двоеточие:

```
p=re.compile(r"((?:abc){2} \d{2})")
```

```
или p=re.compile(r"(?:abc){2} \d{2}")
```

```
['abcabc 23']
```

```
print(p.findall(str))
```

Еще один пример:

```
my="./dumps/backup2011-04-01.txt"
```

```
p=re.compile(r"(backup(20|19)\d{2}-(\d{2}))")
```

```
print(p.findall(my))
```

```
[('backup 2011-04', '20', '11', '04')]
```

Совпадения подстрок возвращаются в кортеж в том порядке, в котором открывающиеся скобки встречаются в шаблоне.

Добавление **?:** позволяет оставить только нужные подстроки:

```
p=re.compile(r"(backup (?:20|19)(?:\d{2})-(?:\d{2}))")
```

```
['backup 2011-04']
```

Еще пример.

```
s="test text"
```

```
p=re.compile(r"([a-z]+((st)|(xt)))",re.S)
```

```
print (p.findall(s))
```

```
[('test', 'st', 'st', ''), ('text', 'xt', '', 'xt')]
```

В результате будет получен список с двумя кортежами, каждый из которых содержит четыре элемента. Все элементы соответствуют фрагментам, заключенным в шаблоне в круглые скобки. Три последних элемента каждого кортежа лишние. Для того чтобы не выводить результат, необходимо добавить символы **?:** после каждой открывающейся скобки подшаблона. В результате полученный список будет включать только фрагменты, соответствующие полному регулярному выражению:

```
p=re.compile(r"([a-z]+(?:st|xt))",re.S)    ['test', 'text']
```

Более интересный и практичный пример:

```
y='он шел, он плыл два дня, тот шел три дня, тот плыл вчера'
```

```
print (y)
```

```
p=re.compile(r"((?:он|тот) (?:шел|плыл))")
```

```
['он шел', 'он плыл', 'тот шел', 'тот плыл']
```

Фрагментам внутри круглых скобок можно дать имена, для этого после открывающейся круглой скобки следует указать комбинацию символов **?P<name>**

Например:

```
email="unicross@gmail.com"
```

```
p=re.compile (r"(?P<name>[a-z0-9_.-]+) @(?P<host> (?:[a-z0-9-]+\.) + [a-z]{2,6})")
```

```
print(p.findall(email))
```

```
r=p.search(email)
```

```
print (r.group ("name"))
```

```
print (r.group("host"))
```

К найденному фрагменту в круглых скобках внутри шаблона можно обратиться с помощью обратной ссылки. Для этого порядковый номер круглых скобок в шаблоне указывается после бек-слеша \. Нумерация скобок внутри шаблона начинается с единицы.

```
print ("механизм обратных ссылок")
```

```
s "<b>Text1</b> Text 2 <I>Text3</I>"
```

```
p=re.compile(r"<([a-z]+)>(.*?)</\1>",re.S|re.I )
```

```
print (p.findall(s))
```

```
механизм обратных ссылок
[('b', 'Text1'), ('I', 'Text3')]
```

Кроме того, внутри круглых скобок могут быть расположены следующие

конструкции:

- **(?=...)** – положительный просмотр вперед. Выводит все подстроки, после которых расположены указанные символы. Например, можно вывести подстроки шаблона, расположенные после запятой или пробела:

```
s="text1, text2, text3 text4"
p=re.compile(r"\w+(?=[, ])")
print(p.findall(s))
```

['text1', 'text2', 'text3']

- **(?!...)** – отрицательный просмотр вперед. Выведет все подстроки, после которых нет указанных символов. Например, выводятся слова, после которых нет запятой:

```
s="text1, text2, text3 text4"
p=re.compile(r"[a-z]+\d(?![,])")
print(p.findall(s))
```

['text3', 'text4']

- **(?<=...)** – положительный просмотр назад. Выводит все подстроки, перед которыми расположены указанные в строке символы. Например, все слова, перед которыми стоит запятая с пробелом.

```
s="text1, text2, text3 text4"
p=re.compile(r"(?<=,)[a-z]+[0-9]", re.S|re.I)
print(p.findall(s))
```

Положительный просмотр назад
['text2', 'text3']

- **(?<!...)** – отрицательный просмотр назад. Выводит все подстроки, перед которыми нет указанных в скобках символов. В примере, слова, перед которыми расположен пробел, но нет запятой.

```
print ("Отрицательный просмотр назад")
p=re.compile(r"(?<!,) \w+\d",re.S|re.I)
print(p.findall(s))
```

Отрицательный просмотр назад
[' text4']

3.5. Методы и функции модуля re

К модулю **re** Python относятся следующие методы:

- **split** — расщепляет строку на подстроки по заданному регулярному выражению;
- **finall** — ищет все шаблоны в строке;
- **match** — проверяет совпадение с шаблоном начала строки;
- **search** — ищет первое совпадение с шаблоном (аналогично методу **match**, но не требует совпадений с началом строки);
- **finditer** — аналогичен функции **findall()**, но возвращает итератор, а не список;
- **sub** — ищет совпадения с шаблоном и заменяет на указанные значения.

В функциях библиотеки **regex** шаблон всегда является первым аргументом функции, а строка — вторым. В зависимости от функции, возвращаемое значение — итератор, новая строка или совпадающий объект. Обычно вместо функции можно использовать одноименный метод строки шаблона, параметром которого является строка.

Метод **match(строка [начальная позиция [, конечная позиция]])** проверяет совпадение шаблона с началом строки. Метод вызывается объектом шаблона, созданного с помощью метода **re.compile**. Если совпадение найдено, то возвращается объект **MatchObject**, в противном случае возвращается значение **None**:

```
p=re.compile(r"\d+")
if (p.match("1230str")):
    print ("Найдено")
else:
    print("Нет")
if (p.match("str1230")):
    print ("Найдено")
else:
    print("Нет")
if (p.match("str1230", 3)):
```

```
метод match
Найдено
Нет
Найдено
```



```

    print("Найдено")
else:
    print("Нет")

```

Вместо метода `match()`, можно использовать одноименную функцию **`re.match(pattern, string [,модификатор])`**.

В качестве **`pattern`** можно использовать строку с регулярным выражением или скомпилированное регулярное выражение. В модификаторе указываются флаги, которые используются функцией **`compile()`**:

```

print (re.match('abc','abcde'))
print(re.match('cde','abccde'))

<_sre.SRE_Match object; span=(0, 3), match='abc'>
None

```

Метод **`search (строка[, начальная позиция [, конечная позиция]])`** похож на `match()`, однако он ищет совпадения с шаблоном не только в начале строки, но и в любой ее части. Если соответствие найдено, то возвращается объект **`MatchObject`**, в противном случае возвращается значение **`None`**. Метод осуществляет поиск по всей строке, но возвращает только первое найденное совпадение:

```

print ("метод search")
if (p.search("str1230")):
    print ("Найдено")
else:
    print("Нет")

```

метод search
Найдено

Аналогично функции **`match()`**, вместо метода **`search()`** можно использовать одноименную функцию **`re.search(pattern, string [,модификатор])`**:

```

print (re.search("\d{2}','de34abc'))

```

<_sre.SRE_Match object; span=(2, 4), match='34'>

Объект **`MatchObject`**, возвращаемый методами (функциями) **`match()`** и **`search()`**, имеет следующие основные свойства и методы:

- **`re`** — ссылка на скомпилированный шаблон, указанный в методах (функциях) `match()` и `search()`. Через эту ссылку доступны следующие свойства:

- **groups** – количество групп в шаблоне;
- **groupindex** – словарь с названиями групп и их номерами.

- **group ([id] или name [..., idN или nameN])** – возвращает фрагменты, соответствующие шаблону. Если параметр не указан, или указано значение 0, то возвращается фрагмент, полностью соответствующий шаблону. Если указан номер или название группы, то возвращается фрагмент, совпадающий с этой группой. Через запятую можно указать несколько номеров или названий групп. Тогда возвращается кортеж, содержащий фрагменты, соответствующие группам. Если нет группы с соответствующим номером или названием, то возбуждается исключение `IndexError`.

- **groupdict ([значение по умолчанию])** — возвращает словарь, содержащий значения именованных групп. С помощью необязательного параметра можно указать значение, которое будет выводиться вместо значения **None** для групп, не имеющих совпадений:

```
print ("Объект MatchObject")
p=re.compile(r"(?P<num>\d+)(?P<str>\w+)")
m=p.search("12345string 67890text")
print(m.re.groups)
print (m.re.groupindex)
print(m.group(), m.group(1),m.group(2))
print(m.group("num"))
print()
print(m.groupdict())
```

```
Объект MatchObject
2
{'num': 1, 'str': 2}
12345string 12345 string
12345
{'num': '12345', 'str': 'string'}
```

- **start ([номер группы или название]), end([номер группы или название]), span([номер группы или название])** возвращает индекс начала (конца) фрагмента (кортеж, содержащий начальный и конечный индексы фрагмента). Если параметр не указан, то фрагментом является полное соответствие с шаблоном, в противном случае — соответствие с указанной группой. Если соответствия нет, то возвращает значение -1 (-1) (-1,-1):

```

m=p.search("67890text")
print(m.re.groups)
print (m.group())
print (m.start(), m.end(),m.span())
print (m.group(1))
print (m.start(1), m.end(1),m.span(1))
print (m.group(2))
print (m.start(2), m.end(2),m.span(2))

```

```

2
67890text
0 9 (0, 9)
67890
0 5 (0, 5)
text
5 9 (5, 9)

```

Метод **findall** (строка [, начальная позиция [, конечная позиция]]) ищет все совпадения с шаблоном. Если соответствие найдено, то он возвращает список фрагментами, в противном случае – пустой список. Если внутри шаблона есть более одной группы, то каждый элемент списка будет кортежем, а не строкой:

```

print("Метод findall")
p=re.compile(r"\d+")
print(p.findall("2011, 2012, двух тысячный, 2018"))
p=re.compile(r"(\d{3})-(\d{2})-(\d{2}))")
print(p.findall("322-22-11, 356-34-12"))

```

```

Метод findall
['2011', '2012', '2018']
[('322-22-11', '322', '22', '11'), ('356-34-12', '356', '34', '12')]

```

Вместо метода **findall()**, так же, как и раньше, можно использовать функцию **findall(): re.findall(шаблон, строка [, модификатор])**.

Метод **finditer** (строка[, начальная позиция [, конечная позиция]]) аналогичен методу **findall()**, но возвращает итератор, а не список. На каждой итерации цикла **for** возвращается один найденный объект соответствия шаблону **MatchObject**:

```

print("Метод finditer")
p= re.compile (r"\d+")
for m in p.finditer("2011, 2012, двух тысячный, 2018"):

```

```
print (m.group(0), "start:", m.start(), "end:",m.end())
```

Вместо метода **finditer ()** можно использовать функцию **finditer()**:

re.finditer (шаблон, строка [, модификатор])

```
print("Функция finditer")
```

```
p= re.compile (r"<b>(.*?)</b>", re.I|re.S)
```

```
s="<b>Text1</b> Text2 <b>Text3</b>"
```

```
print (p.findall(s))
```

```
for m in re.finditer(p,s):
```

```
    print (m.group(1))
```

```
Функция finditer  
['Text1', 'Text3']  
Text1  
Text3
```

В данном примере **group(1)** выделяет группу символов, соответствующую стоящему в скобках шаблону, а **(.+?)** делает квантификатор нежадным.

Метод **sub (новый фрагмент или ссылка на функцию, строка для замены [, максимальное количество замен])** ищет все совпадения с шаблонами и заменяет их на указанные значения. Если совпадения не найдены, то возвращается исходная строка. Внутри нового фрагмента можно использовать рассмотренные ранее обратные ссылки **\номер** и **\g <название>**, соответствующие группам внутри шаблона:

```
print ("метод sub - замена")
```

```
p=re.compile(r"<(P<tag1>\w+)><(P<tag2>[a-z]+)>")
```

```
print(p.sub(r"<\2><\1>","<br><hr>"))
```

#\номер

```
print (p.sub(r"<\g<tag2>><\g<tag1>>","<br><hr>"))
```

#\g <название>

```
метод sub - замена  
<hr><br>  
<hr><br>
```

В качестве первого параметра функции можно указывать ссылку на функцию (имя функции без скобок). Данная функция в качестве параметра будет использовать объект **MatchObject**, соответствующий найденному по шаблону фрагменту. Результат, возвращаемый такой функцией, является фрагментом для замены в функции **sub**.

В приведенном ниже примере осуществляется увеличение на 10 всех найденных в скобках чисел:

```
#метод sub – замена, с функцией
def repl(m):
    """Функция для замены m- MatchObject"""
    x=int(m.group(0))
    x+=10
    return ("%s" %x)
p=re.compile(r"\d+")
print (p.sub(repl, "2008, 2009,2010,2011, 2012"))
print (p.sub(repl, "2008, 2009,2010,2011, 2012", 2))
```

```
метод sub – замена, с функцией
2018, 2019,2020,2021, 2022
2018, 2019,2010,2011, 2012
```

Вместо метода **sub()** можно использовать функцию **sub()**:

re.sub(шаблон, новый фрагмент или ссылка на функцию, строка для замены [, максимальное количество замен])

```
def repll(m):
    tag1=m.group("tag1").upper()
    tag2=m.group("tag2").upper()
    return "<%s><%s>" % (tag2,tag1)
p=r"<(P<tag1>\w+)><(P<tag2>\w+)>"
print(re.sub(p,repll, "<br><hr>"))
```

```
<HR><BR>
```

Метод **subn()** аналогичен методу **sub()**, только он возвращает не строку, а кортеж из двух элементов — измененной строки и количества произведенных замен:

subn (новый фрагмент или ссылка на функцию, строка для замены [, максимальное количество замен])

```
#метод subn – замена, с функцией
print("метод subn – замена")
```

```
p=re.compile(r"200[79]")
print(p.subn("0","2007,2009,2010,2011"))

метод subn - замена
('0,0,2010,2011', 2)
```

Метод **split (Исходная строка[,Лимит])** разбивает строку по шаблону и возвращает список подстрок. Если вторым параметром задано число, то в списке будет указанное количество подстрок. Если подстрок больше указанного количества, то список будет содержать еще один элемент, который будет содержать остаток строки:

```
print("метод split - разбиение строки")
p=re.compile(r"[\s,.]+" ) # любой пробел, точка или запятая
print(p.split("word1, word2\nword3\rword4.word5"))
print(p.split("word1, word2\nword3\rword4.word5",2))

метод split - разбиение строки
['word1', 'word2', 'word3', 'word4', 'word5']
['word1', 'word2', 'word3\rword4.word5']
```

Если разделитель не найден в строке, то список будет содержать только один элемент, содержащий исходную строку. Вместо метода можно использовать функцию **split ()**.

Контрольные вопросы и практические задания по теме «Регулярные выражения»

1. Используя регулярные выражения, напишите программу, суммирующую произвольное количество введенных пользователем целых чисел. Программа завершает работу и осуществляет суммирование при вводе слова *stop*. Программа допускает ввод отрицательного числа. Если введена строка вместо числа, пользователю выдается сообщение: «Необходимо ввести число, а не строку».

2. Попросите пользователя ввести фамилию и имя. Используя регулярные выражения, поменяйте местами имя и фамилию и выведите их на консоль.

3. Используя регулярные выражения, выведите на консоль все слова, стоящие после открывающейся кавычки во фрагменте текста.

4. Проверьте *e-mail*, введенный пользователем, на соответствие шаблону. Присвойте имена группе, соответствующей имени домена, и группе, соответствующей имени пользователя. Если введенная строка не соответствует шаблону, выведите пользователю сообщение. Используя метод *group* объекта *MatchObject*, выведите введенный электронный адрес, имя домена и имя пользователя на консоль. Выведите на консоль запись в обратном порядке: вначале имя домена, затем имя пользователя.

5. В текстовом фрагменте выберите слова, стоящие в начале предложения (после точки и пробела).

6. Проверьте строку, введенную пользователем, на соответствие шаблону, включающему число от двух до пяти цифр и наименование валюты (\$, *Euro*, *UAH*, *USD*, *SEK*, *NOK*, *RUB*). Присвойте имя группе шаблона, соответствующей числу, и группе, соответствующей имени валюты. Если введенная строка не представляет собой число от двух до пяти цифр включительно, с наименованием валюты, выведите пользователю соответствующее сообщение. Используя метод *group()* объекта *MatchObject*, выведите введенную строку, число и наименование валюты на консоль. Выведите на консоль запись в обратном порядке: вначале валюта, затем число.

7. В текстовом фрагменте выберите слова, стоящие в конце предложения (перед точкой и пробелом).

8. Попросите пользователя ввести дату, в форме число (от 1 до 31), наименование месяца и четыре цифры текущего года. Проверьте введенную дату на соответствие шаблону. Присвойте имя группе шаблона, соответствующей числу и группе, соответствующей месяцу. Если введенная строка не соответствует введенному шаблону, выведите соответствующее сообщение. Используя метод *group()* объекта *MatchObject* выведите введенную строку, число и месяц на консоль. Выведите на консоль запись в обратном порядке: вначале месяц, затем число и год.

9. В текстовом фрагменте выберите слова, не стоящие после чисел (число и пробел).

10. Попросите пользователя ввести дату, в форме число и наименование месяца. Проверьте дату на попадание в первую декаду месяца (первые 10 дней). Присвойте имя группе шаблона, соответствующей числу и группе, соответствующей месяцу. Если введенная строка не соответствует введенному шаблону, выведите соответствующее сообщение. Используя метод `group()` объекта `MatchObject`, выведите введенную строку, число и месяц на консоль. Выведите на консоль запись в обратном порядке: вначале месяц, затем число.

11. В текстовом фрагменте выберите слова, не стоящие перед числами (пробел и число).

12. Попросите пользователя ввести свое научное звание (*профессор* или *доцент*), фамилию и инициалы. Например: *профессор Иванов С. И.* Проверьте введенную строку на соответствие шаблону. Присвойте имя группе шаблона, соответствующей фамилии, и группе, соответствующей инициалам. Если введенная строка не соответствует введенному шаблону, выведите соответствующее сообщение. Используя метод `group()` объекта `MatchObject`, выведите введенную строку, фамилию и инициалы пользователя на консоль. Выведите на консоль запись в обратном порядке: вначале инициалы, а затем фамилия.

13. Попросите пользователя ввести полное доменное имя, которое должно начинаться с наименования протокола “`http://`” или “`https://`”, затем перечислите домены различных уровней, состоящие из букв и разделенные точкой, заканчивающиеся последним двухбуквенным или трехбуквенным доменом верхнего уровня (зоной); в конце может быть необязательный слеш (`http://example.com/`). Присвойте имя группе шаблона, соответствующей домену верхнего уровня. Если введенная строка не соответствует введенному шаблону, выведите соответствующее сообщение. Используя метод `group()` объекта `MatchObject`, выведите введенную строку и имя домена верхнего уровня на консоль. Выведите на консоль запись в обратном порядке: вначале домен верхнего уровня, а затем наименование протокола.

14. Попросите пользователя ввести четырехбайтовый IP-адрес (IPv4) в десятичном представлении. С помощью шаблона регулярного выражения проверьте правильность введенной строки. Четырехбайтный адрес представляет собой четыре числа от 0 до 255, разделенных точками <https://ru.wikipedia.org/wiki/IPv4>. Проверьте введенную строку на соответствие шаблону. Присвойте имя группам шаблона, представляющим первый, второй, третий и четвертый байты. Если введенная строка не соответствует введенному шаблону, выведите соответствующее сообщение. Используя метод *group()* объекта *MatchObject*, выведите введенную строку и значения каждого из четырех байтов на консоль. Выведите на консоль запись в обратном порядке: вначале четвертый байт адреса, затем третий, второй и первый, разделенные точкой.

Тема 4. НОРМАЛИЗАЦИЯ ТЕКСТА

4.1. Сегментация слов.

4.2. Нормализация.

4.3. Стемминг.

4.4. Сегментация предложений в тексте.

4.5. Токенизация в NLTK Python.

4.6. Использование регулярных выражений для стемминга в Python.

4.7. Встроенные стеммеры пакета NLTK.

4.8. Лемматизатор пакета NLTK.

Контрольные вопросы и практические задания по теме «Нормализация текста».

4.1. Сегментация слов

Любая обработка текста должна включать его нормализацию. Процесс нормализации текста включает:

- сегментацию или токенизацию слов в тексте (Segmenting/tokenizing);
- нормализацию слов (приведение их к нормальной/канонической) форме;
- сегментацию предложений в тексте.

Первый этап обработки текстов – токенизация – используется в качестве предварительного этапа части-речной разметки (POS-tagging), при поиске общих слов, удалении нежелательных слов, подобных общим или повторяющимся словам и т. д.

Токен представляет собой последовательность буквенных и/или цифровых символов, отделенную слева и справа знаками форматирования текста и/или знаками препинания. Разбивка текста на токены называется **токенизацией**. Токены обычно совпадают со словами, поэтому термину “токен” обычно соответствует слово в теоретической лингвистике.

Существует разграничение между уникальными токенами и общими токенами. **Уникальный токен** (иногда называемый **типом**) обозначает токен без учета количества его повторений в тексте, а термин “**общие токены**” – обозначает ко-

личество токенов с учетом их частотности. Например, Shakespeare corpus имеет объем около 884 тысячи слов, 31 тысяча – уникальные токены. Switchboard corpus телефонных разговоров, содержащий 2500 разговоров около 500 человек, включает 2,4 миллиона общих токенов, из которых только 20 000 токенов – это уникальные токены.

Токенизация представляет собой сложную задачу, часто не имеющую единственного решения. Выбор и определение токенов (это число со знаком доллара, это просто число и знак доллара отдельно, это строка до точки или числа, выделенные точками – как дата) всегда зависят от области применения.

Выделим несколько традиционно возникающих проблем автоматической токенизации:

1. Разделения текста по пробелам может привести к разрыву токена, который следует рассматривать как цельный. Одним токеном должны быть распознаны такие словосочетания, как географические названия, личные имена, сокращения, устойчивые словосочетания и т. п. Например: “*New York*”, “*N.A.T.O.*”, “*San Francisco*” и др.

2. В ряде языков существует проблема разделения апострофом. Например, в английском языке, апостроф, определяющий притяжательный падеж, не выделяет отдельное слово (*Finland's capital*), тогда как в примерах “*I'm*”, “*What're*” апостроф разделяет два слова. Во французском языке апостроф может использоваться для сокращения определенного артикля перед словом.

3. Во многих языках существует проблема обработки слов, содержащих дефис. Например, в английском языке разделение дефисом используется для разных целей: для объединения существительных в названиях (*Hewlett-Packard*) и для разделения гласных букв в словах (*co-education*), а также для группирования слов (*state-of-the-art*). В данном примере, “*co-education*” “*Hewlett-Packard*” рассматривается как один токен, “*state-of-the-art*” – как несколько токенов, а случай “*Hewlett-Packard*” может быть рассмотрен по-разному.

4. В немецком языке слова и соответственно токены могут записываться слитно (*Computerlinguistik*).

5. В восточно-азиатских языках вообще нет пробелов.

4.2. Нормализация

Следующим этапом автоматической обработки текста, наступающим после того как слова выделены, т. е. текст сегментирован на токены, эти токены необходимо нормализовать. Понятие нормализации слов имеет несколько значений; в общем случае **нормализация** – это приведение слова к начальной (канонической) форме. Например: *Кошками* -> *кошка*; *Бежал* -> *бежать*.

Лемматизация в общем случае также представляет собой нормализацию. **Лемма** — это начальная (словарная) форма слова, вместе с информацией о форме построения слова и его части речи. Под лемматизацией в лингвистике понимают процесс группировки всех изменяемых форм слова для их последующего анализа в виде одного элемента (леммы). Слова относятся к одной и той же лемме, если они имеют одинаковую основу, относятся к одной и той же части речи и имеют почти один и тот же смысл слова. Например, *cat* и *cats* – это различные формы одного и того же слова. В компьютерной лингвистике лемматизация представляет собой процесс выделения леммы для данного слова. Например, в английском языке глагол “*to open*” может проявляться как “*opened*”, “*opens*”, “*opening*”. Базовая или каноническая форма “*open*”, которая представлена в словаре, называется леммой данного слова. Осуществляется лемматизация на основе алгоритмов морфологического анализа, она связана с распознаванием элементов морфологической структуры слова – корня, основы, суффикса, окончания.

Процесс лемматизации представляет собой трудную задачу, включающую понимание контекста, определение части речи слова в предложении, базирующиеся на знании грамматики данного языка.

И хотя процесс лемматизации тесно связан с процессом стемминга, стемминг не требует знаний контекста, поэтому он не может различать слова, которые имеют различные значения, в зависимости от того, к какой части речи они относятся. Поэтому обычно стеммеры легче имплементировать, и они работают быстрее, чем лемматизаторы, тогда как их меньшая точность не сильно влияет на работу некоторых приложений NLP.

Этап нормализации обычно включается в такие приложения NLP, как информационный поиск, машинные перевод и другие. Например, в информационном поиске лемматизация позволяет расширить класс искомых терминов. Это особенно актуально для флективных языков. При использовании в поиске словосочетания “*банковых систем*”, пользователь, обычно, подразумевает включение таких форм как “*банковая система*”, “*банковой системы*” и других. В системе машинного перевода определение канонической формы слова необходимо для поиска заголовка словарной статьи перевода.

4.3. Стемминг

С нормализацией связан стемминг (*Stemming*), который также является частью процесса нормализации текста. **Стемминг** – это процесс нахождения основы слова по заданному исходному слову. Цель стемминга отождествить основы различных словоформ, имеющих одно значение. Например, в результате стемминга все такие слова, как *automate(s)*, *automatic*, *automation*, будут приведены к одной основе *automat*.

Наиболее часто стемминг применяется в системах информационного поиска для расширения возможности поиска совпадений множеств слов запроса и документов, позволяя существенно повысить показатели полноты и точности поиска в информационно-поисковых системах (ИПС). Например, если при информационном поиске запрашивается слово “*applications*”, то поисковая машина будет осуществлять поиск документов, содержащих как слово “*application*”, так и “*applications*”.

Конкретный способ решения задачи поиска основы слов называется “**алгоритм стемминга**”, а конкретная реализация – “**стеммер**” (*stem* – основа).

На входе стеммера подается список токенов, на выходе получают список их стемм, который не всегда совпадает с четкой основой, определенной теоретической лингвистикой. Термин “**стемма**” в NLP обозначает последовательность символов, остающуюся после удаления части строк и выполняющую функцию отождествления токенов. Стемминг, таким образом, представляет собой упрощённую форму лемматизации, где мы просто отсекаем суффиксы.

Наиболее простой алгоритм, который обычно используется для стемминга английского языка – алгоритм Портера или *стеммер Портера*.

Согласно данному алгоритму:

- 1) создается база данных словообразующих суффиксов и список простых правил;
- 2) на первом шаге отсекается наибольший возможный суффикс и проверяется, удовлетворяет ли оставшаяся основа заданным правилам;
- 3) если оставшаяся основа не удовлетворяет заданным правилам, то убирается следующий суффикс.

Пример наиболее простых правил замены суффиксов для английского языка:

sses → ss	(caresses → caress)
ies → i	(ponies → poni)
ss → ss	(caress → caress)
s → 0	(cats → cat)
ationat → ate	(relational → relate)
izer → ize	(digitizer → digitize)
ator → ate	(operator → operate)
al → 0	revival → reviv
able → 0	adjustable → adjust
ate → 0	activate → activ

В приведенном ниже стеммере используются более сложные правила отсечения суффиксов. Например, для суффиксов **-ing -ed** действует такое правило: если после отсечения в слове не остается ни одной гласной буквы, то такое отсечение неправильное:

(*v*)ing →	(walking → walk)
0	
	Sing → sing
(*v*)ed → 0	(plastered → plaster)

Данное правило не подтверждается только у слов, подобных *nothing, something and anything*.

Алгоритм Портера имеет следующие недостатки:

- он часто обрезает слово больше необходимого, что затрудняет получение правильной основы слова, например: *кровать*->*крова* (при этом реально неизменяемая часть – *кровать*, но стеммер выбирает для удаления наиболее длинную морфему);
- он не справляется со всевозможными изменениями корня слова (например, с выпадающими и беглыми гласными);

Здесь приведены примеры для английского языка; флективные и агглютинативные языки имеют значительно более сложные морфологии и требуют более сложных алгоритмов стемминга.

4.4. Сегментация предложений в тексте

Несмотря на кажущуюся простоту, разбить текст на предложения достаточно сложно. Очевидно, если предложение заканчивается на ! или ?, то его можно выделить достаточно однозначно. Однако если встречается точка, то это может быть как граница предложения, так и аббревиатура, сокращение (*Dr., N.A.T.O.*) или число (*.02%* или *4.3*).

Для выделения предложений может быть использован бинарный классификатор, который может быть построен с помощью:

- правил;
- регулярных выражений;
- классификатора, на базе машинного обучения (*machine learning classifiers*);
- дерева принятия решений и других подходов.

Одним из наиболее простых методов решения является построение дерева принятия решений, которое используется в простом алгоритме *if-then-else* и имеет общий вид, показанный на рис. 4.1.

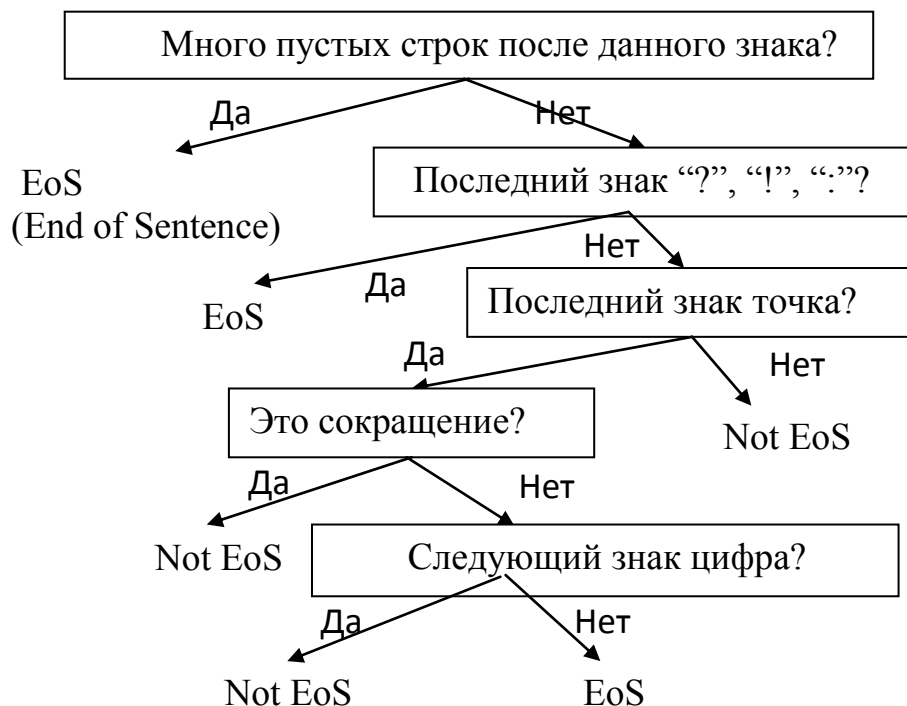


Рисунок 4.1 – Общий вид дерева решений, используемого при сегментации предложений

4.5. Токенизация в NLTK Python

Обычно для токенизации используют модуль **nltk.tokenize** библиотеки NLTK и регулярные выражения.

Функция **word_tokenize** данного модуля позволяет разбивать предложения на слова и знаки пунктуации. Функция возвращает список токенов:

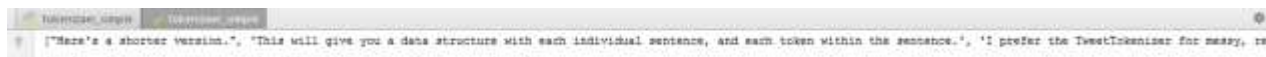
```
from nltk.tokenize import word_tokenize
print (word_tokenize("Hi there!"))
```

```
tokenizaer_simple > tokeni
['Hi', 'there', '!']
```

Для того чтобы разделить текст на предложения, можно использовать функцию **sent_tokenize** модуля **nltk.tokenize**. Функция возвращает список, каждым элементом которого является предложение:

```
from nltk.tokenize import sent_tokenize
{print (sent_tokenize("Here's a shorter version. This will give you a data structure \
With each individual sentence, and each token " within the sentence. \
```


I prefer the TweetTokenizer for messy, real world language. ")))}



Класс **TweetTokenizer()** модуля разработан для гибкого и легкого использования в новых областях и задачах, связанных с обработкой твитов – сообщений, включающих большое количество неалфавитных символов. Конструктор класса использует три параметра по умолчанию:

TweetTokenizer(preserve_case=True, reduce_len=False, strip_handles=False).

Если сохранить все значения параметров по умолчанию, то в этом случае конкатенация полученных токенов даст исходную строку.

Параметр **preserve_case** по умолчанию имеет значение **True**. Если изменить его на **False**, то все буквы в токенах будут прописные.

Параметр **strip_handles** по умолчанию имеет значение **False**. Если изменить его значение на **True**, то токены, начинающиеся с символа **@** будут игнорироваться.

Параметр **reduce_len** по умолчанию имеет значение **False**. Если изменить его значение на **True**, многократное повторение букв будет игнорироваться.

Метод **tokenize(text)** данного класса, принимая в качестве параметра строку текста, возвращает список токенов:

```
from nltk.tokenize import TweetTokenizer
s0=" '@remy: This is waaaaayyyy Too much for YOU!!!!!!' #dfk ###ldld"
tknzs=TweetTokenizer()
print(tknzs.tokenize(s0))
print()
tknzs=TweetTokenizer(preserve_case=False)
print(tknzs.tokenize(s0))
print()
tknzs=TweetTokenizer(strip_handles=True, reduce_len=True )
```

```
print(tknzr.tokenize(s0))

tknzs=TweetTokenizer(strip_handles=True)

print(tknzs.tokenize(s0))
```

```
TweetTokeniz
↑ [ '"', '@remy', ':', 'This', 'is', 'waaaaayyyy', 'Too', 'much', 'for', 'YOU', '!', '!', '!', '!', '#dfk', '###ldld' ]
↓ [ '"', '@remy', ':', 'this', 'is', 'waaaaayyyy', 'too', 'much', 'for', 'you', '!', '!', '!', '!', '#dfk', '###ldld' ]
Process finished with exit code 0
```

Работа функции **regex_tokenize()** модуля **nlTK.tokenize** аналогична работе функции **re.findall()** пакета **regex**. Однако функция **nlTK.regex_tokenize()** является более эффективной для задачи токенизации и позволяет избегать особого обращения работы с круглыми скобками.

Например, для того чтобы выделить в качестве токенов слова с дефисами и аббревиатуры, достаточно написать следующий фрагмент программного кода:

```
from nlTK.tokenize import regex_tokenize

text="That U.S.A. and USA poster-print costs $12.40 at New-York... $145.9%"

print (text)

pattern= r"(?:[A-Z]\.)+|[A-Z]+\s|\w+-\w+"

print (regex_tokenize(text, pattern))
```

```
TweetTokeniz
↑ That U.S.A. and USA poster-print costs $12.40 at New-York... $145.9%
↓ ['U.S.A.', 'USA ', 'poster-print', 'New-York']
```

4.6. Использование регулярных выражений для стемминга в Python

Существуют различные способы программно отделить основу от слова. Наиболее простым способом для английского языка является отсекаание от слова любого суффикса:

```
def stem(word):

    for suffix in ['ing', 'ly', 'ed', 'ious', 'ies', 'ive', 'es', 's', 'ment']:
```

```

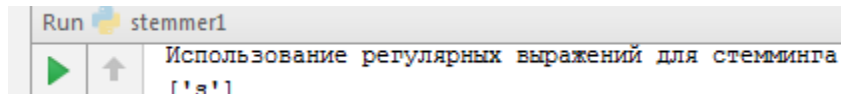
if word.endswith(suffix):
    return word[:-len(suffix)]
return word

```

Кроме того, для стемминга можно использовать регулярные выражения. Используя регулярное выражение, можно:

1) выделить только суффикс:

```
print(re.findall(r"^(.*(ing|s|ly|ed|ious|ive|es|s|ment))$", "applications"))
```



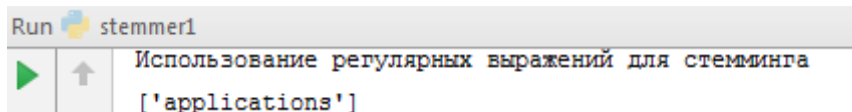
```

Run stemmer1
Использование регулярных выражений для стемминга
['s']

```

2) выделить только слово, содержащее один из заданных суффиксов:

```
print(re.findall(r"^(.*(?:ing|s|ly|ed|ious|ive|es|s|ment))$", "applications"))
```



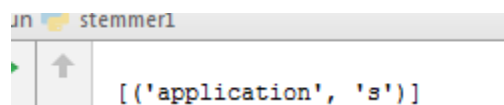
```

Run stemmer1
Использование регулярных выражений для стемминга
['applications']

```

3) разделить слово на основу и суффикс и выделить кортеж, включающий основу и суффикс:

```
print(re.findall(r"^(.*)(ing|s|ly|ed|ious|ive|es|s|ment)$", "applications" ))
```



```

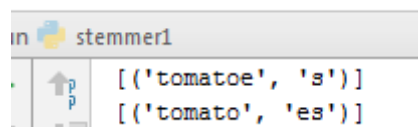
Run stemmer1
Использование регулярных выражений для стемминга
[('application', 's')]

```

Для того чтобы анализировать слова, содержащие суффикс - es , следует сделать квантификатор * в подстроке .* “нежадным”:

```
print(re.findall(r"^(.*)(ing|s|ly|ed|ious|ive|es|s|ment)$", "tomatoes" ))
```

```
print(re.findall(r"^(.*?)(ing|s|ly|ed|ious|ive|es|s|ment)$", "tomatoes" ))
```



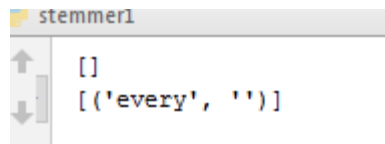
```

Run stemmer1
Использование регулярных выражений для стемминга
[('tomatoe', 's')]
[('tomato', 'es')]

```

Для того чтобы данное регулярное выражение распознавало слова, которые не имеют описанных суффиксов, следует добавить после перечисленных суффиксов квантификатор ?:

```
print(re.findall(r"^(.*?)(ing|s|ly|ed|ious|ive|es|s|ment)$","every" ))
print(re.findall(r"^(.*?)(ing|s|ly|ed|ious|ive|es|s|ment)?$","every" ))
```



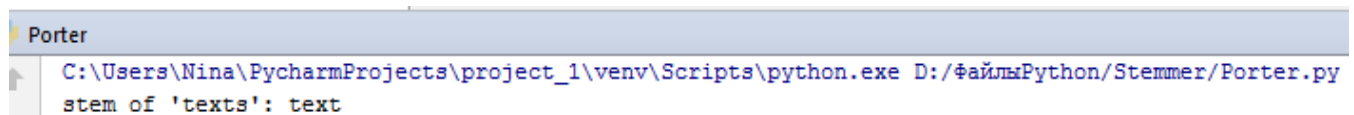
4.7. Встроенные стеммеры пакета NLTK

Библиотека NLTK включает несколько встроенных стеммеров, использовать которые предпочтительней, чем создавать собственные регулярные выражения. Такие стеммеры, как *Porter Stemmer*, *Lancaster stemmer* и *Snowball Stemmer*, обрабатывают большое количество нерегулярных случаев и исключений. Например, *Porter stemmer* корректно обрабатывает такие слова как *lying*, правильно выделяя в качестве основы “*lie*”. Использовать эти стеммеры в NLTK достаточно просто.

Porter Stemmer реализует алгоритм, описанный Мартином Портером (Martin Porter) в 1980 году. Алгоритм представляет собой процесс удаления общих морфологических и флективных окончаний слов английского языка. Его основное использование — нормализация терминов, используемая в препроцессорной обработке систем информационного поиска.

Для использования библиотеки стеммера Портера необходимо подключить модуль **PorterStemmer** пакета **nltk.stem**, создать объект класса **PorterStemmer** данного модуля и использовать метод **stem()** созданного объекта:

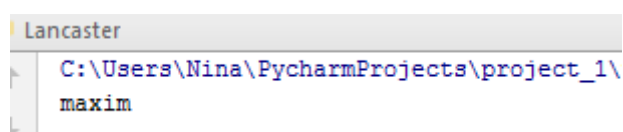
```
from nltk.stem import PorterStemmer
ps=PorterStemmer()
print ("stem of 'texts': " + ps.stem("texts"))
```



Lancaster Stemmer базируется на алгоритме Ланкастер Стеммер (1990), который “укорачивает слово как можно короче”. Использование данного алгоритма может быть особенно выгодным, например, для обработки очень больших текстовых наборов данных.

Использование *Lancaster Stemmer* аналогично использованию Стеммера Портера:

```
from nltk.stem.lancaster import LancasterStemmer
st=LancasterStemmer()
print(st.stem("maximum"))
```



Lancaster
C:\Users\Nina\PycharmProjects\project_1\
maxim

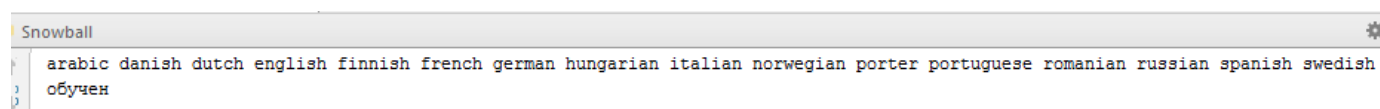
Snowball stemmer представляет собой новую улучшенную версию алгоритма Портера <http://snowball.tartarus.org/>. Практически данный стеммер представляет собой фреймворк для создания алгоритмов стемминга, улучшенных стеммеров английского языка и некоторых других языков.

В отличие от модулей двух предыдущих стеммеров, при создании объекта класса SnowballStemmer необходимо указать в качестве параметра язык стеммера.

```
from nltk.stem import SnowballStemmer
snowball_stemmer=SnowballStemmer("english")
print(snowball_stemmer.stem("going"))
```

Список языков, поддерживаемых **Snowball Stemmer**, можно получить, используя свойство *SnowballStemmer.languages*:

```
print(" ".join(SnowballStemmer.languages))
snowball_st=SnowballStemmer("russian")
print(snowball_st.stem("обучение"))
```



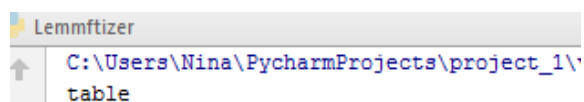
Snowball
arabic danish dutch english finnish french german hungarian italian norwegian porter portuguese romanian russian spanish swedish
обучен

4.8. Лемматизатор пакета NLTK

NLTK-лемматизатор базируется на морфологической функции лексической базы данных английского языка *WordNet* <https://wordnet.princeton.edu/>. *WordNet* напоминает тезаурус, поскольку он группирует слова вместе на основе их значений. Структура WordNet делает его полезным инструментом в вычислительной лингвистике и NLP.

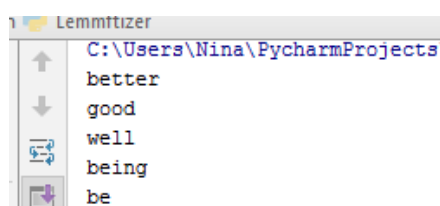
Для того чтобы осуществить лемматизацию английского слова, необходимо подключить модуль *WordNetLemmatizer* пакета *nltk.stem*, создать объект класса *WordNetLemmatizer* и применять для данного объекта метод *lemmatize* (*str* [, *pos='n'*]), используя в качестве параметра слово, подлежащее лемматизации:

```
from nltk.stem import WordNetLemmatizer
wn_l=WordNetLemmatizer()
print (wn_l.lemmatize("tables"))
```



Метод *lemmatize*(*str* [, *pos='n'*]) имеет два аргумента, первый из которых показывает лемматизируемое слово, а второй – используемую при анализе часть речи. Часть речи может быть существительным *pos = 'n'* (используемым по умолчанию), глаголом *pos = 'v'*, прилагательным *pos = 'a'* или наречием *pos = 'r'*:

```
from nltk.stem import WordNetLemmatizer
wn_l=WordNetLemmatizer()
print (wn_l.lemmatize("better"))
print (wn_l.lemmatize("better", pos='a'))
print (wn_l.lemmatize("better", pos='r'))
print (wn_l.lemmatize("being", pos='n'))
print (wn_l.lemmatize("being", pos='v'))
```



Контрольные вопросы и практические задания по теме «Нормализация текста».

1. Существующие приложения Natural Language Processing (NLP).
2. Сегментация слов. Проблемы токенизации слов в тексте.
3. Сегментация предложений в тексте. Бинарный классификатор выделения предложений.
4. Нормализация и лемматизация слов.
5. Стемминг текста. Особенности использования стеммера Портера
6. Токенизация слов с помощью библиотеки nltk python. Функции модуля nltk.tokenize.
7. Использование регулярных выражений для выделения токенов.
8. Создайте текстовый файл, содержащий текст:

With a solid plan and time for revision, most writing tasks can be completed with ease? This course is for those interested in improving their professional writing. Learn how to improve your writing's organization, logic, and style so that whatever kind of writing your work requires, you can get your point [across] eloquently and quickly! This COURSE : will delve into the mechanics of the writing process: identifying an audience, choosing the best structure, and revising early drafts of your work. You will build confidence as you practice planning documents; using [organizational] [strategies] such headings and subheadings, and finding misused words and proofreading errors.

9. Разбейте текст на предложения.
10. Разделите на токены второе и третье предложения.
11. Создайте словарь уникальных токенов текста.
12. С помощью созданного регулярного выражения найдите первую подстроку, написанную в скобках. Выведите ее на экран.
13. В четвертом выделенном предложении найдите и выведите на экран подстроку, записанную большими буквами и пробелами, после которых стоит двоеточие.
14. Скачайте любой понравившийся текст с сайта <http://www.gutenberg.org/catalog/> в файл text.txt. Все токены файла text.txt запи-

шите маленькими буквами, оставив только буквенные токены, создайте словарь уникальных токенов, отсортируйте его. Запишите код с использованием технологии «List Comprehensions» (включение списка).

15. Рассмотрите различные варианты токенизации приведенного ниже текста. Опишите их особенности, по-разному определяя токены. В последнем варианте выделите только буквенные слова:

" 'Kinto by Mozilla - An open source Parse alternative >>
<https://github.com/Kinto/kinto/> #python #parse OOOOOOOO!!! @kindl!"

16. Используя приведенный ниже список твитов, выделите с помощью функции **regex_tokenizer()** в первом твите (и последнем твите) токены с хэш-тегами, т. е. начинающиеся с символа # или @ (подобно #Data или @text):

[“ #True_store In the ZONE: The Official Music Video, Jut Now @example”,

“Katie #UTE @B0TTersnike. By far your bestalbum since Won’t Go Quietly @example back to your best”,

”@realDonaldTrump FLORIDA – it is imperative that you heed the directions of your State and Local Officials. Please be prepared, be careful and be SAFE! #HurricaneMichael ready gov”,

“@NWSTallahassee 8am Intermediate Advisory from @NHC_Atlantic upgrades #HurricaneMichael into a category 2 hurricane.”]

17. Используя экземпляр класса **TweetTokenizer**, выведите на экран токены всех твитов. Запишите код с использованием технологии «List Comprehensions» (включение списка).

18. Используя функцию **regex_tokenizer()** выделите из строки "That U.S.A. and USA poster-print costs \$12.40 at New-York... \$145.9%" (или любой другой подобной строки текста) токены, представляющие обычные слова, аббревиатуры, слова с дефисом, валюту (вещественное число с точкой, начинающееся со знака \$ и возможно имеющее знак % в конце числа) и многоточие (...) одновременно.

Тема 5. POS TAGGING (МОРФОЛОГИЧЕСКАЯ РАЗМЕТКА)

5.1. Задачи POS-тегирования.

5.2. Функция `pos_tag()` пакета `nltk`.

5.3. Инсталляция пакета `rumorphy2`.

5.4. `Rumorphy2` – морфологический анализатор для русского и украинского языков.

Контрольные вопросы и практические задания по теме «POS Tagging (морфологическая разметка)»

5.1. Задачи POS-тегирования

Процесс классификации слов по частям речи и их соответствующая разметка называется частиречной разметкой или **POS tagging** (Part-of-Speech tagging). Иногда данный процесс POS-тегирования еще называют **word classes** или **lexical categories**. POS tagging представляет собой способ автоматического морфологического анализа, когда каждая словоформа входной фразы рассматривается изолированно, вне связей с другими словами предложения. В результате такого анализа каждому слову в тексте (корпусе) приписывается метка или тег, обозначающие часть речи и грамматические характеристики данного слова.

Первым размеченным корпусом стал **Brown Corpus**, который был морфологически размечен в 1971 году двумя студентами-магистрами. В результате данного студенческого эксперимента размеченный Brown Corpus стал использоваться во многих тысячах исследований по всему миру.

Для осуществления автоматического морфологического тегирования, необходимо:

- 1) выбрать множество тегов (**tagset**), которое будет использоваться для решения конкретной задачи разметки;
- 2) определить, какой именно тег из множества тегов должен быть применен к каждому конкретному токenu текста, множество которых получено на предыдущем этапе токенизации текста.

Tagset — это множество тегов или категорий слов, используемых для задачи грамматического тегирования. Такое множество меток первого размеченного

Brown Corpus включало 77 различных меток классов-слов для английского языка. Такое большое количество классов определяло не только главные части речи (*noun, verb, article, adverb, preposition, conjunction, participle, pronoun*), но и значения различных подклассов, таких, как *singular noun and plural noun, positive adjectives, comparative adjectives and superlative adjectives* и т. д.

На сегодня существует несколько tagset для морфологического POS-тегинга текстов английского языка. Например, текущий стандарт C7 tagset, используемый для разметки British National Corpus (BNC), включает 130 тегов. Наиболее популярной системой POS-тегов английского языка является множество **tagset** размеченного **Penn Treebank** корпуса¹, которое используют, например, **NLTK default tagger** и **Stanford CoreNLP tagger**. Фрагмент данного тегсета показан в табл. 5.1.

Таблица 5.1 – Фрагмент обозначения POS-тегов, используемого для размеченного Penn Treebank корпуса

Тег	Значение	Пример
JJ	adjective or numeral, ordinal	new, good, high, special, big, fourth
RB	Adverb	really, already, still, early, now
DT	determiner	the, a, some, most, every, no
MD	modal verb	will, can, would, may, must, should
NN	noun, singular or mass	year, home, time, education
NNP	proper noun singular or mass	Alison, Africa, April, Washington
CD	Cardinal number	twenty-four,, 1991, 14:24
PRP	Personal pronoun	he, their, her, its, my, I, us
IN	preposition	on, of, at, with, by, into, under
TO	the word to	to
UH	interjection	ah, bang, ha, whee, hmpf, oops

¹ https://www.ling.upenn.edu/courses/Fall_2003/ling001/penn_treebank_pos.html

Продолжение табл. 5.1

Тег	Значение	Пример
VB	Verb, base form	is, has, get, do, make, see, run
VBZ	verb, present tense, 3rd person singular	is, bases reconstructs marks mixes displeases seals
VBD	past tense	said, took, told, made, asked
VBG	Verb, gerund or present participle	making, going, playing, working
VBN	past participle	given, taken, begun, sung

В современных корпусах используется вертикальный, горизонтальный или псевдо-XML стили разметки.

Пример POS Tagging разбора двух английских предложений “*This is a short sentence. So is this.*”:

```

<sentence id="0">
  <word wid="0" pos="DT">This</word>
  <word wid="1" pos="VBZ">is</word>
  <word wid="2" pos="DT">a</word>
  <word wid="3" pos="JJ">short</word>
  <word wid="4" pos="NN">sentence</word>
  <word wid="5" pos=".">.</word>
</sentence>
<sentence id="1">
  <word wid="0" pos="RB">So</word>
  <word wid="1" pos="VBZ">is</word>
  <word wid="2" pos="DT">this</word>
  <word wid="3" pos=".">.</word>
</sentence>

```

Так как морфология языков различна, поэтому и значения тегов морфологического анализа различны для разных естественных языков. Например, так будет выглядеть псевдо-XML стиль морфологического разбора предложения «*Вот так, за пять минут до съемок, родился новый персонаж*»:

```
<se> <w><ana lex="вот" gr="PART"></ana>Вот</w>
<w><ana lex="так" gr="ADV-PRO"></ana>так</w>,
<w><ana lex="за" gr="PR"></ana>з`а</w>
<w><ana lex="пять" gr="NUM=acc"></ana>пять</w>
<w><ana lex="минута" gr="S,f,inan=pl,gen"></ana>мин`ут</w>
<w><ana lex="до" gr="PR"></ana>до</w>
<w><ana lex="съемка" gr="S,f,inan=pl,gen"></ana>съёмок</w> ,
<w><ana lex="родиться" gr="V,pf,intr,med=m,sg,praet,indic">
</ana>род`илс`я</w>
<w> <ana lex="новый" gr="A=m,sg,nom,plen"> </ana>н`овый</w>
<w><ana lex="персонаж" gr="S,m,anim=sg,nom"></ana>персон`аж</w>.
</se>.
```

В данном примере используется морфологический разбор национального корпуса русского языка², где:

- предложения заключены в теги <se>;
- внутри предложений расположены слова в теге <w>;
- информация о каждом слове содержится в теге <ana>;
- значение атрибута lex показывает лексему слова;
- значения атрибута gr представляют собой множество грамматических категорий данного слова, первая из которых показывает часть речи, а последующие представляют значения грамматических категории данной части речи (число, падеж, время глагола и т.д.).

Например, в табл. 5.2 показаны некоторые из общепринятых обозначений грамматических категорий русского языка.

² <http://www.ruscorpora.ru/>

Таблица 5.2 – Обозначения грамматических категорий русского языка

Тег	Значение	Пример
S	существительное	стол, лингвистика
A	прилагательное	красивый, походный
NUM	количественное числительное	семь, восемьдесят
A-NUM	порядковое числительное	седьмой, восьмидесятый
V	глагол	идти, пришел
ADV	наречие	пешком, красиво
PARENTH	вводное слово	кстати, по-моему
S-PRO	местоим. сущ.	она, что
A-PRO	местоим. прил.	который, твой
ADV-PRO	местоим. нареч.	где, вот
PRAEDIC-PRO	местоим. предик.	некого, нечего
PR	предлог	в, на, из
CONJ	союз	и, или, но
PART	частица	бы, же, пусть
'INTJ':	'межд.'	увы, ох

5.2. Автоматическое тегирование средствами пакета nltk. Функция `pos_tag()`

Для тегирования можно использовать регулярные выражения, т. е. определять необходимый тег токену на основе соответствия шаблону.

Класс `nltk.RegexpTagger(regexps, [backoff=None])` обозначает токены тегами, сравнивая их с заданными регулярными выражениями. Параметр `regexps (list(tuple(str, str)))` представляет собой список двузначных кортежей (`regexp`, `tag`), первый элемент которого представляет собой шаблон регулярного

выражения, а второй – значение тега, соответствующего данному регулярному выражению. Пары (**regexp, tag**) применяются в том порядке, в котором они написаны. Если слово не соответствует ни одному из заданных регулярных выражений, то используется значение, заданное вторым необязательным параметром *backoff*; по умолчанию используется обозначение *None*.

Метод *tag(tokens)* данного класса возвращает список кортежей, состоящих из двух элементов. Первый элемент кортежа представляет собой токен, а второй – тег, определенный классом **RegexTagger**:

```
import nltk

text="We will also see how tagging is the second step \
in the typical NLP pipeline, following tokenization."

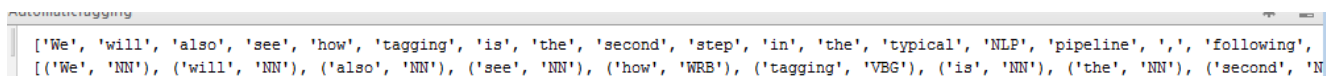
# разделение текста на токены:
word_list=nltk.word_tokenize(text)

print(word_list)

#список шаблонов и соответствующих им тегов:
patterns=[(r'.*ing$', 'VBG'), ('how', 'WRB'), (r'.*', 'NN')]

# создание объекта размеченного согласно шаблонам текста:
regex_tagger=nltk.RegexpTagger(patterns)

# получение списка кортежей размеченных токенов:
print (regex_tagger.tag(word_list))
```



```
[('We', 'NN'), ('will', 'NN'), ('also', 'NN'), ('see', 'NN'), ('how', 'WRB'), ('tagging', 'VBG'), ('is', 'NN'), ('the', 'NN'), ('second', 'N
```

В приведенном выше примере последнее регулярное выражение *“.*”* означает, что все остальные, не описанные предыдущими шаблонами, токены будут обозначены метками *<NN>*, что эквивалентно тегированию по умолчанию.

При вероятностном автоматическом тегировании каждый токен помечается наиболее вероятным тегом. Если вероятность обозначения слова невозможно определить, то можно использовать **тегирование по умолчанию**. Такое тегирование обычно предполагает, что большинство слов являются существительными. Если разметить все токены достаточно большого текста наиболее вероятным те-

гом– <NN>, то правильно будет размечено около 13 % токенов.

Для POS tagging фрагмента текста используется функция **pos_tag()** пакета **nlk**:

```
import nltk
text=nltk.word_tokenize ("And now for something completely different")
print (text)
print(nltk.pos_tag(text))
```

В полученной разметке: CC – coordinating conjunction, RB – adverbs, IN – preposition, NN – noun, JJ – adjective.

Для того чтобы получить информацию о значении конкретного тега, можно использовать функцию **upenn_brown_tagset (имя_тега)** модуля **help** пакета **nlk**:

```
nltk.help.upenn_tagset ('NN')
```

Получить список размеченных токенов текстов, состоящих их нескольких предложений, можно с помощью следующего программного кода:

```
print("POS_tagging на уровне предложения")
print()
#разбивка текста на предложения
text=nltk.sent_tokenize ("Durable solutions are one of the best \
investments in peace and stability. Bilateral differences are best \
resolved bilaterally. We fully concur that this issue is best \
addressed through a consensus text. ")
print (text)
```

```

#создание списка списков токенов каждого предложения
text_word_tokens=[nltk.word_tokenize(sentence_token) for sentence_token in
text]

print (text_word_tokens)

#создание списка списков POS-тегированных токенов каждого предложения
text_tagged=nltk.pos_tag_sents(text_word_tokens)

print(text_tagged)

#информация о POS-тегах слова "best"

nltk.help.upenn_tagset ('JJS')
nltk.help.upenn_tagset ('RB')
nltk.help.upenn_tagset ('RBS')

```

Один и тот же токен (в приведенном выше примере слово “best”) может быть размечен различными тегами, то есть может относиться к различным частям речи. POS-тегирование в большинстве случаев корректно определяет необходимый для данного слова тег, базируясь на контексте предложения:

```

POS_tagging на уровне предложения

['Durable solutions are one of the best investments in peace and stability.', 'Bilateral differences are best resolved bilaterally.', 'We fu

[['Durable', 'solutions', 'are', 'one', 'of', 'the', 'best', 'investments', 'in', 'peace', 'and', 'stability', '.'], ['Bilateral', 'differen

[[('Durable', 'JJ'), ('solutions', 'NNS'), ('are', 'VBP'), ('one', 'CD'), ('of', 'IN'), ('the', 'DT'), ('best', 'JJS'), ('investments', 'NNS
JJS: adjective, superlative
    calmest cheapest choicest classiest cleanest clearest closest commonest
    corniest costliest crassest creepiest crudest cutest darkest deadliest
    dearest deepest densest dinkiest ...
RB: adverb
    occasionally unabatingly maddeningly adventurously professedly
    stirringly prominently technologically magisterially predominately
    swiftly fiscally pitilessly ...
RBS: adverb, superlative
    best biggest bluntest earliest farthest first furthest hardest
    heartiest highest largest least less most nearest second tightest worst

```

Тег, который добавляется к слову при морфологической разметке, зависит не только от самого слова, но и от контекста его использования в предложении. По этой причине автоматическое тегирование осуществляется на уровне предложений скорее, чем на уровне списка слов.

5.3. Установка пакета pymorphy2

Пакет **pymorphy2**³ представляет собой морфологический анализатор для русского языка (с экспериментальной версией для украинского языка), написанный на языке Python и использующий словари из OpenCorpora⁴.

Для того чтобы его использовать для морфологического анализа русского языка, прежде всего, необходимо установить библиотеку либо с помощью утилиты **pip**:

```
pip install pymorphy2,
```

либо воспользовавшись инструментарием PyCharm.

После этого необходимо аналогичным способом установить отдельно пакет словаря:

```
pip install -U pymorphy2-dicts-ru
```

Для того чтобы иметь возможность работать с морфологическим анализатором украинского языка, необходимо в командном режиме установить пакет с **github.com**:

```
pip install --https://github.com/kmike/pymorphy2/archive/master.zip#egg=pymorphy2
```



После чего необходимо установить словари украинского языка:

```
pip install -U pymorphy2-dicts-uk
```

5.4. Pymorphy2 – морфологический анализатор для русского и украинского языков

Для осуществления морфологического анализа русского языка необходимо создать объект класса **MorphAnalyzer()**. Для осуществления морфологического

³ <https://pymorphy2.readthedocs.io/en/0.2/user/index.html>

⁴ <http://opencorpora.org/dict.php>

анализа украинского языка при создании объекта класса **MorphAnalyzer** необходимо использовать параметр **lang='uk'**:

```
import pymorphy2

morph=pymorphy2.MorphAnalyzer()

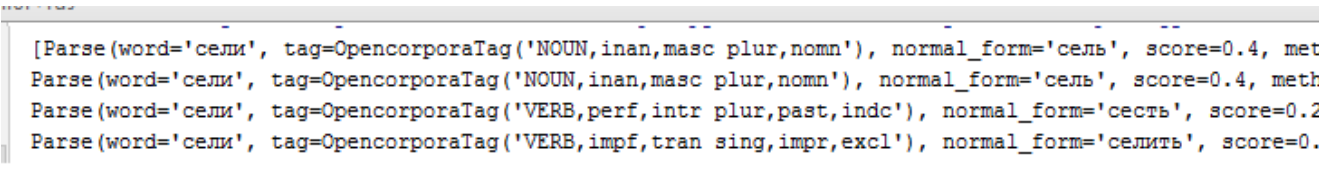
morph_ukr=pymorphy2.MorphAnalyzer(lang='uk')
```

Обратите внимание, что экземпляры класса **MorphAnalyzer** обычно занимают порядка 15Мб оперативной памяти так как они загружают в память словари, данные для предсказателя и т. д.

С помощью метода **parse()** созданного объекта **MorphAnalyzer** можно разобрать отдельное слово и получить один из возможных вариантов разбора. **MorphAnalyzer.parse()** возвращает список, включающий возможные варианты разбора данного слова.

Например, слово “*цели*” в первом разборе представлено существительным множественного числа от слова “*цель*”, а во втором разборе рассматривается как глагол, нормальная форма которого “*сесть*”; в четвертом разборе это слово представляет собой глагол повелительного наклонения, от глагола “*селить*”:

```
print(morph.parse('цели'))
gr=morph.parse('цели')[0]
print(gr)
gr=morph.parse('цели')[1]
print(gr)
gr=morph.parse('цели')[3]
print(gr)
```

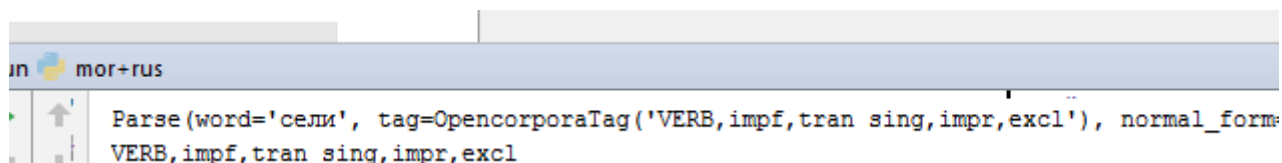


```
[Parse(word='цели', tag=OpencorporaTag('NOUN,inan,masc plur,nomn'), normal_form='цель', score=0.4, met
Parse(word='цели', tag=OpencorporaTag('NOUN,inan,masc plur,nomn'), normal_form='цель', score=0.4, meth
Parse(word='цели', tag=OpencorporaTag('VERB,perf,intr plur,past,indc'), normal_form='сесть', score=0.2
Parse(word='цели', tag=OpencorporaTag('VERB,impf,tran sing,impr,excl'), normal_form='селить', score=0.
```

Свойства полученного морфологического разбора позволяют получить из него тег морфологической разметки, нормальную форму слова и граммемы.

Для того чтобы получить множество морфологических характеристик данного варианта разбора или его тег разметки, необходимо использовать свойство **tag** данного разбора. Например, для четвертого разбора слова “*цели*”[3] тег представляет собой *'VERB,impf, tran sing,impr,excl'*:

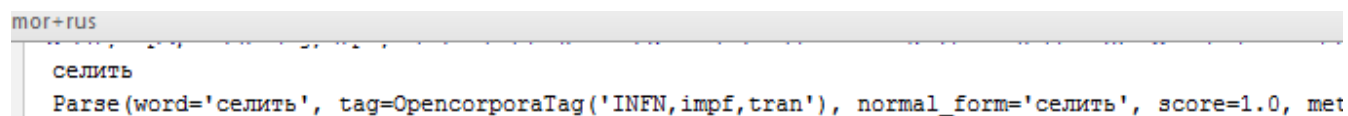
```
gr=morph.parse('цели')[3]
print(gr)
print(gr.tag)
```



The screenshot shows a terminal window with the title 'mor+rus'. The command executed is `Parse(word='цели', tag=OpencorporaTag('VERB,impf,tran sing,impr,excl'), normal_form=VERB,impf,tran sing,impr,excl)`. The output is `VERB,impf,tran sing,impr,excl`.

Нормальную форму слова можно получить с помощью свойств **normal_form** или **normalized** заданного морфологического разбора:

```
gr=morph.parse('цели')[3]
print (gr.normal_form)
print (gr.normalized)
```



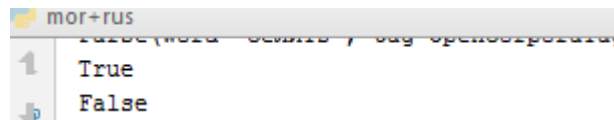
The screenshot shows a terminal window with the title 'mor+rus'. The command executed is `Parse(word='цели', tag=OpencorporaTag('INFN,impf,tran'), normal_form='целить', score=1.0, met)`. The output is `целить`.

Любые морфологические характеристики, которые могут быть включены в тег, называются **граммемами**. Например, **VERB** – глагол (личная форма), *impf* – несовершенный вид глагола, *tran* – переходный глагол, *sing* – единственное число, *impr* – повелительное наклонение, *excl* – невключенность говорящего в действие. Все доступные при разборе граммемы⁵ показаны в **Приложении А**

Для того чтобы проверить, есть ли в данном теге отдельная граммема, можно использовать оператор **in**:

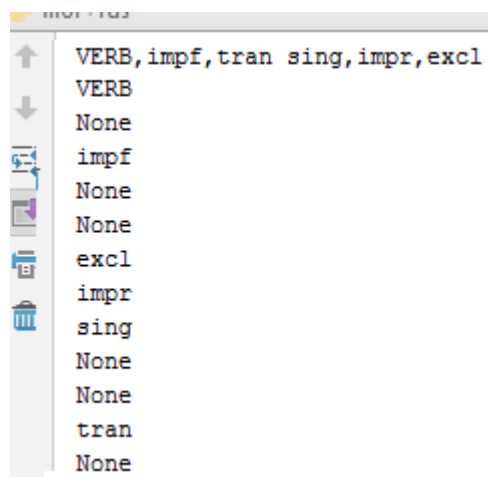
```
print('VERB' in gr.tag)
print ({'NOUN', 'plur'} in gr.tag )
```

⁵ <http://opencorpora.org/dict.php?act=gram>



У каждого тега есть атрибуты или свойства, с помощью которых можно получить значения определенных грамматических категорий, например, времени, лица, числа и др. Если запрашиваемая характеристика для данного тега не определена, то возвращается *None*:

```
gr=morph.parse('цели')[3]
print(gr.tag)
print(gr.tag.POS)
print(gr.tag.animacy) # одушевлённость
print(gr.tag.aspect)  # совершенный и несовершенный вид
print(gr.tag.case)    # падеж
print(gr.tag.gender)  # род
print(gr.tag.involvement) # включенность говорящего в действие
print(gr.tag.mood)    # наклонение (повелительное, изъявительное)
print(gr.tag.number)  # число
print(gr.tag.person)  # лицо
print(gr.tag.tense)   # время
print(gr.tag.transitivity) # переходность
print(gr.tag.voice)   # залог (действительный, страдательный)
```



Теги и граммы в `pymorphy2` записываются латиницей (например, `NOUN`). Но часто удобнее использовать кириллические названия грамм (например, `СУЩ` вместо `NOUN`). Чтобы получить тег в виде строки, записанной кириллицей, используется свойство `cyr_repr`:

```
gr=morph.parse('стали')[0]
print(gr.tag)
print(gr.tag.cyr_repr)
```

`Pymorphy2` позволяет склонять слова, т. е. ставить их в заданную форму. Чтобы просклонять слово, нужно сначала разобрать слово, выбрать из предложенных вариантов разбора правильный или нужный и затем использовать метод `inflect()`, указав в качестве параметра нужную форму слова. В примере, приведенном ниже, – это родительный падеж существительного:

```
morph=pymorphy2.MorphAnalyzer()
gr=morph.parse('ученый')[0]
print(gr.inflect({'gent'}))
```

```
mor+rb2_2
Parse(word='учёного', tag=OpencorporaTag('NOUN,anim,masc sing,gent'), normal_form='учённый', score=1.0,
```

Кроме того, используя свойство `lexeme`, можно получить лексему слова выбранного варианта разбора.


```
print(gr.lexeme)
```

`Pymorphy2` возвращает все допустимые варианты разбора, но на практике обычно нужен только один вариант, тот, который является правильным. В общем случае для выбора правильного варианта разбора необходимо рассматривать не только само слово, но и связь слов в предложении. Для анализа слова на уровне морфологии, без учета его синтаксиса и семантики, используется такой параметр разбора `Pymorphy2`, как `score`:

```
mor+rus
[Parse(word='цели', tag=OpencorporaTag('NOUN,inan,masc plur,nomn'), normal_form='цель', score=0.4, method:
Parse(word='цели', tag=OpencorporaTag('NOUN,inan,masc plur,nomn'), normal_form='цель', score=0.4, method:
Parse(word='цели', tag=OpencorporaTag('VERB,perf,intr plur,past,indc'), normal_form='сесть', score=0.2, m
Parse(word='цели', tag=OpencorporaTag('NOUN,inan,masc plur,nomn'), normal_form='цель', score=0.4, method:
```

Score – это вероятностная оценка $P(\text{tag}|\text{word})$, показывающая условную вероятность того, что данное слово размечено именно этим тегом (имеет именно этот вариант разбора). Получить это значение можно с помощью свойства **score**:

```
print(gr.score)
```



```
0.818181
```

Условная вероятность $P(\text{tag}|\text{word})$ оценивается на основе корпуса OpenCorpora. Для этого определяются все неоднозначные слова со снятой неоднозначностью. Для каждого слова вычисляется количество раз сопоставления слова с конкретным тегом. Затем, на основе полученных частот, вычисляется условная вероятность тега при условии данного слова (с использованием сглаживания Лапласа).

На сегодня оценка вероятности $P(\text{tag}|\text{word})$ осуществляется на основе OpenCorpora примерно для 20 тыс слов. Для тех слов, у которых такой оценки нет, вероятность $P(\text{tag}|\text{word})$ либо считается равномерной (для словарных слов), либо оценивается на основе эмпирических правил (для несловарных слов).

Все разборы слова, представляемые пакетом `rumorphy2`, сортируются по убыванию рассчитанной вероятности их появления (**score**), поэтому везде в примерах берется первый вариант разбора из всех имеющихся.

Все остальные варианты разбора могут использоваться в каких-то особенных задачах. Например, если нужно просклонять слово и если известно, что на входе ожидается слово в именительном падеже, то лучше брать вариант разбора в именительном падеже, а не первый.

Оценка **score** для анализа украинских текстов к моменту данной публикации не доступна.

Контрольные вопросы и практические задания по теме «POS Tagging (морфологическая разметка)»

1. Результат автоматического морфологического анализа.
2. POS-tagging английского языка.
3. Особенности инсталляции пакета морфологического анализа украинского языка `rumorphy2`?
4. Особенности создания объекта класса `MorphAnalyzer()` для украинского языка?
5. Каким методом объекта класса `MorphAnalyzer()` можно получить морфологический разбор заданного слова? Что является параметром данного метода? Что собой представляет морфологический разбор, возвращаемый данным методом?
6. Какими свойствами и методами обладает полученный морфологический разбор?
7. Как получить тег морфологического разбора? Что собой представляет полученный тег?
8. Как получить нормальную форму слова?
9. Что подразумевается под понятием “граммема” в `rumorphy2`? Как получить значение грамлемы для конкретного морфологического разбора?
10. Как получить кириллические названия грамлем?
11. Для чего используется свойство грамматического разбора **score**?
12. Как определяется параметр **score** в морфологическом разборе `rumorphy2`?
13. Осуществить POS-tagging приведенного ниже фрагмента текста:

Machine Learning evolved from computer science that primarily studies the design of algorithms that can learn from experience. To learn, they need data that has certain attributes based on which the algorithms try to find some meaningful predictive patterns. Majorly, ML tasks can be categorized as concept learning, clustering, predictive modeling, etc. The ultimate goal of ML algorithms is to be able to take decisions with-

out any human intervention correctly. Predicting the stocks or weather are a couple of applications of machine learning algorithms.

14. Используя модуль **help** пакета nltk, получить информацию о метках CC, IN, NN и JJ.

15. Разработать регулярные выражения для морфологической разметки некоторых слов украинских или русских текстов. Осуществить POS-tagging фрагмента текста, соответствующего языка.

16. Показать варианты морфологического разбора для русского слова «блестящий».

17. В цикле для каждого морфологического разбора вывести на экран:

- a) тег;
- b) нормальную форму;
- c) POS;
- d) записать тег кириллицей;
- e) дать вероятностную оценку выбора именно этого варианта разбора.

18. Показать варианты морфологического разбора, рассматривающего слова «блестящий» как прилагательное:

- a) вывести значения морфологических категорий данного слова, свойственных прилагательному;
- b) проверить существование в данном разборе морфологических категорий, свойственных причастию.

19. Изменить разбор слова, поставив его в творительный падеж множественного числа

20. Вывести полную лексему данного слова, рассмотрев его в качестве прилагательного и в качестве причастия.

21. Осуществить морфологическую разметку любого небольшого текстового файла с русскоязычным текстом. Размеченный только тегами текст сохранить в другом файле.

22. Повторить задания 1–6 для любого украинского слова и любого текста на украинском языке.

Тема 6. СИНТАКСИЧЕСКИЙ АНАЛИЗ

6.1. Сущность синтаксического анализа текста.

6.2. Деревья синтаксической зависимости.

6.3. Проективные деревья синтаксической зависимости.

6.4. Система составляющих.

6.5. Использование контекстно-свободных грамматик для реализации размеченных систем составляющих.

Контрольные вопросы и практические задания по теме «Строки Python».

6.1. Сущность синтаксического анализа текста

Синтаксический анализ в общем случае происходит в три этапа:

- 1) установление синтаксических связей между словоформами в предложении (контекстный анализ), проводимое на уровне словосочетаний;
- 2) построение формализованной структуры синтаксических отношений во фразе, проводимое на уровне предложения;
- 3) построение межфразовой синтаксической структуры, проводимой на уровне дискурса.

На вход **синтаксического анализа (СА)** предложения поступают:

- выходные данные морфологического анализа;
- и правила синтаксиса языка.

Результаты POS Tagging не отражают структурных связей между элементами фразы (т. е. словами предложения). В то же время слова в предложении не могут следовать в произвольном порядке и передавать при этом некоторую мысль. Каждый язык имеет свой собственный синтаксис. Целью СА является построение синтаксической структуры предложений входного текста

Теоретическую лингвистику в синтаксисе языков больше интересуют сравнение грамматик естественных языков и синтаксические законы, применимые одновременно к большому количеству разных языков, то есть типология.

Целью компьютерной лингвистики в области синтаксиса является построение автоматизированного анализатора или синтаксического парсера (parser) от-

дельного языка. Этот анализатор должен уметь:

- выделять простые предложения в составе сложного;
- устанавливать связи между словами;
- строить полное синтаксическое дерево предложения, т. е. разбирать предложение в категориях синтаксиса (подлежащее, сказуемое, прямое дополнение и др.).

В прикладной лингвистике наиболее употребляемыми являются два способа описания правил синтаксиса предложений естественного языка, т. е. два способа формального описания синтаксической структуры предложения:

- 1) деревья синтаксического подчинения (синтаксической зависимости);
- 2) система составляющих;

Выбор используемого формального представления в большой мере зависит от языка, для которого строится модель синтаксиса.

6.2. Деревья синтаксической зависимости

Деревья синтаксической зависимости (или деревья синтаксического подчинения) основываются на теории зависимости, в соответствии с которой между словоформами во фразе устанавливаются отношения подчинения. Одна словоформа предложения считается независимой, все остальные зависят от нее прямо или косвенно. За исключением независимой словоформы, каждая словоформа имеет только одного «хозяина», от которого она непосредственно зависит.

Диаграмму связей между словоформами во фразе можно представить в виде дерева. В теории графов **дерево** – это связный ориентированный граф, не содержащий циклов и кратных ребер. Для каждой пары его вершин существует единственная цепь, соединяющая их. **Корень дерева** – это вершина графа, из которой выходят управляющие стрелки, но в нее не входит ни одна из стрелок.

Исследование большого числа предложений показало, что связи между словами во фразе практически всегда образуют древовидную структуру, называемую деревом синтаксического подчинения, в узлах которого стоят словоформы. Зависимые слова являются модификаторами или аргументами тех слов, от которых

они зависят. Например, дерево зависимости предложения *Three small children are eating their lunches under the tree* показано на рис. 6.1.

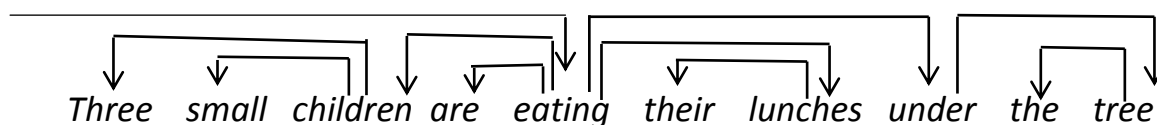


Рисунок 6.1 – Пример дерева зависимости

Корнем рассматриваемого предложения является глагол «eating». Это центральное слово предложения, и иногда это указывается в модели с помощью дуги зависимости, поступающей от края и указывающей главное слово предложения. С ним связаны четыре его аргумента: «children», «are», «lunches» и «under». В свою очередь слово «children» имеет аргументы «Three» и «small». Слово «are» не имеет модификаторов или аргументов, а слово «lunches» имеет модификатор «their». Предлог «under», как всегда, имеет дополнение; его дополнением является аргумент *tree*, модификатором которого в свою очередь является *the*. Множество стрелок на рис. 6.1 показывает анализ зависимостей в предложении.

Дерево зависимости можно представить не линейно, а в двухмерном изображении (рис. 6.2).

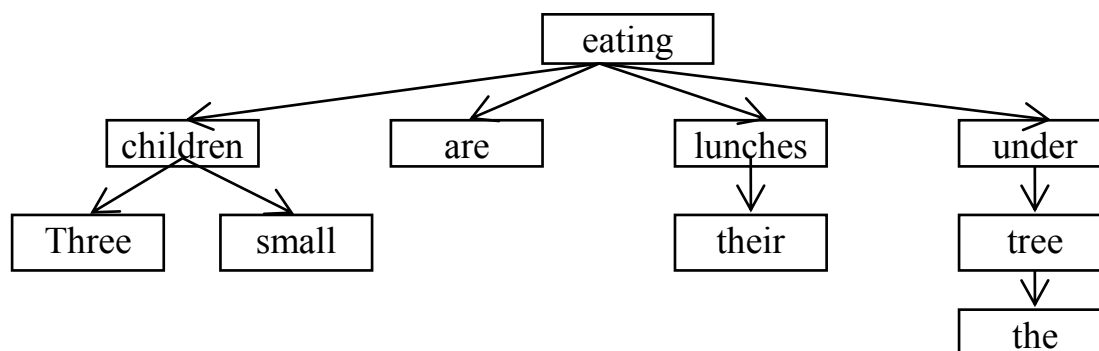


Рисунок 6.2 – Пример дерева зависимости в двухмерном изображении

Если синтаксическая структура фразы включает также указания о типе каждой связи, дерево становится размеченным (рис. 6.3).

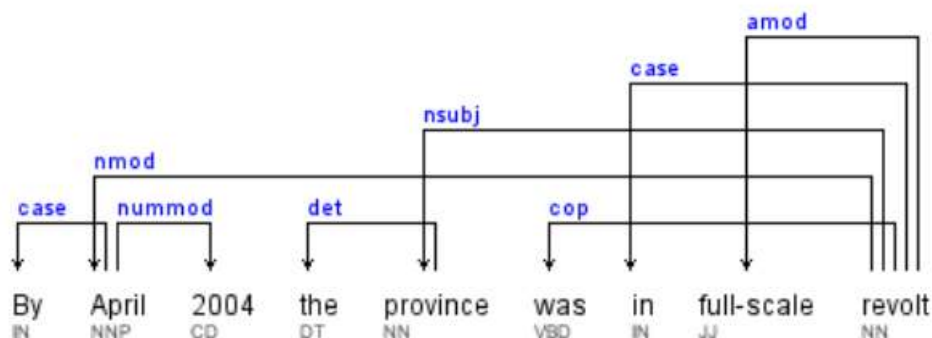


Рисунок 6.3 – Пример размеченного дерева зависимости, построенного Stanford Universal Dependencies parser⁶

Записывают такие деревья синтаксической зависимости чаще всего в виде XML- документов (рис. 6.4).

```

</tokens>
<dependencies type="basic-dependencies">
  <dep type="root">
    <governor idx="0">ROOT</governor>
    <dependent idx="9">described</dependent>
  </dep>
  <dep type="advmod">
    <governor idx="9">described</governor>
    <dependent idx="1">however</dependent>
  </dep>
  <dep type="punct">
    <governor idx="9">described</governor>
    <dependent idx="2">,</dependent>
  </dep>
  <dep type="det">
    <governor idx="8">official</governor>
    <dependent idx="3">a</dependent>
  </dep>
  ..
  ...

```

Рисунок 6.4 – Пример размеченного дерева зависимости, записанного в формате XML-документа

Обычно выделяют следующие виды связи:

- пред – предикативная связь между подлежащим и сказуемым;
- опред – отношение между именем существительным и согласованным с ним определением;
- обст – связь между глаголом и обстоятельством;
- упр – связь между глаголом и управляемым им существительным;

⁶ <https://nlp.stanford.edu/software/stanford-dependencies.shtml>

- инстр – связь между сказуемым и существительным в творительном падеже, обозначающем имя деятеля;
 - атр – атрибутивное отношение между именем существительным и его несогласованным определением;
 - объект – связь между сказуемым и дополнением;
- и другие типы синтаксических отношений.

Если одному предложению соответствует несколько различных деревьев синтаксической зависимости или несколько размеченных деревьев синтаксической зависимости, то можно говорить о синтаксической омонимии. Пример синтаксической омонимии показан на рис. 6.5.

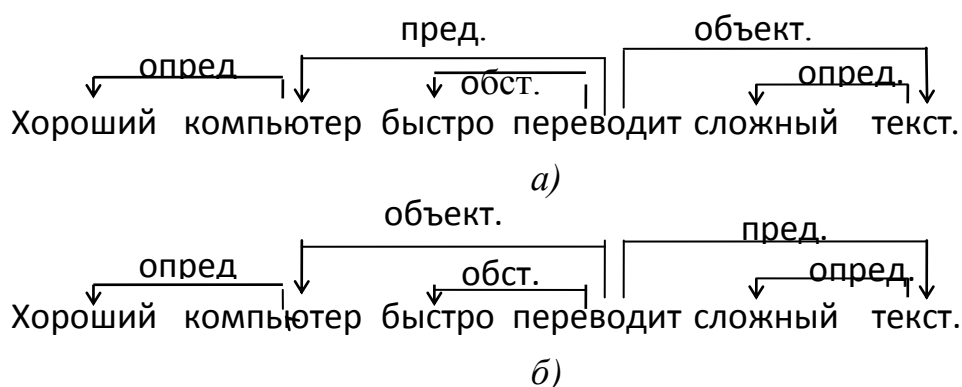


Рисунок 6.5 – Пример синтаксической омонимии предложения

Оба варианта синтаксического анализа *а, б)* предложения «Хороший компьютер быстро переводит сложный текст» полностью соответствуют морфологической информации словоформ и их расположению во фразе. Каждый из двух вариантов размеченных деревьев синтаксической зависимости соответствует разным смыслам. В этом случае, как и во многих других, синтаксическую омонимию можно устранить только с учетом семантики (значения) словоформ.

6.3. Проективные деревья синтаксической зависимости

В классе деревьев зависимости выделяют подкласс проективных деревьев, который описывает подавляющее большинство предложений естественных языков. Требование проективности синтаксической структуры предложения универсально для большинства предложений прозы индоевропейских языков.

Термин «**проективность**», введенный лингвистом И. Лесерфом в 1960 г., означает, что две синтаксически связанные словоформы могут разделяться только словоформами, зависящими прямо или косвенно от одной из них, т. е. синтаксически связанные слова близко расположены в тексте.

В теории графов **проективными деревьями** называются деревья, у которых дуги не пересекаются, а корень не лежит ни под одной из дуг. На рис. 6.6 показан пример проективного дерева синтаксической зависимости.

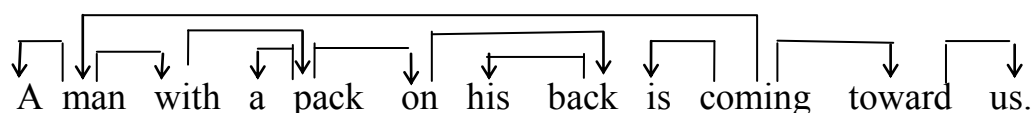


Рисунок 6.6 – Пример проективного дерева синтаксической зависимости

Степень свободы линейного порядка слов в предложении определяется сложностью морфологии языка. Падежные окончания флективных языков (русс., укр., бел.) позволяют свободно менять порядок слов во фразе, сохраняя смысл исходного высказывания. При этом в большинстве предложений письменного языка соблюдается закон проективности.

При автоматическом синтаксическом анализе фраз естественного языка анализируют, как правило, только проективные предложения. Это позволяет во много раз сократить работу алгоритма по определению структуры фразы. Например, если не учитывать проективность, то для фразы в двадцать словоформ можно построить $5 \cdot 10^{24}$ различных синтаксических деревьев. При применении правила проективности число потенциальных деревьев снижается до $4 \cdot 10^{13}$.

Алгоритм, использующий свойство проективности, работает следующим образом. Если одна словоформа отделена от другой только словоформами, зависящими от любой из двух анализируемых, то считается, что первая из этих двух словоформ **прецедентна** второй. Синтаксическое соответствие при этом следует проверять не для всех пар слов во фразе, а только для пар с прецедентностью.

Таким образом, проверка на проективность является фильтром, позволяющим отбросить немалую часть неправильных структур.

Формализация синтаксических структур по методу деревьев подчинения имеет ряд недостатков. Деревья синтаксической зависимости не позволяют передавать структуру предложений с вводными словами-обращениями, вводными оборотами. Кроме того, такая структура показывает только связи между отдельными словами, тогда как связи между словосочетаниями не описываются.

6.4. Система составляющих

Второй способ моделирования синтаксической структуры предложения естественного языка – **система составляющих**, которая организует слова в гнездовые составляющие. Системы составляющих выделяют во фразе две равнозначные группы словоформ: группу подлежащего и группу сказуемого.

Данный подход основан на той идее, что предложение можно последовательно раскладывать на все более мелкие отрезки, непосредственно связанные между собой. Отношения между словами указываются путем расстановки во фразе скобок так, чтобы каждая пара скобок охватывала связанные друг с другом элементы (такими элементами могут быть отдельные слова или группы слов; ранее заключенные в скобки отрезки), называемые составляющими.

Например, синтаксический разбор фразы «Fed raises interest rates» можно представить следующим образом:

(Fed (raises (interest rates))).

Понятно, что составляющие представляют собой некоторые единицы предложения. Это слова и само предложение в целом. Существует несколько подходов, используемых для того, чтобы определить, какие еще единицы предложения являются составляющими. Например, составляющими являются словосочетания, которые остаются вместе при изменении синтаксической конструкции фразы. Вторым подходом является возможность замещения и расширения составляющей предложения.

Исходя из последнего подхода в предложении «I sat on the box», можно расширить составляющую «on the box», и получить фразу «I sat right on top of the box». Или вообще заменить ее. Получатся следующие составляющие:

I sat (on the box).

I sat (right on top of the box).

I sat (there).

Существуют и другие лингвистические тесты выделения составляющих в предложении.

Таким образом, множество *M* составляющих образует систему составляющих, если *M* удовлетворяет следующим условиям:

- в *M* входят в качестве элементов как сама фраза целиком, так и все словоформы, взятые отдельно;
- любые две составляющие не пересекаются, либо одна составляющая полностью содержится в другой.

Аналогично деревьям синтаксической зависимости, если у предложения существует несколько возможных синтаксических структур (то есть, возможно разное толкование данного предложения), то оно имеет несколько «правильных» систем составляющих, выражающих наличие синтаксической омонимии.

Следующий пример показывает два разных смысла предложения и соответственно два возможных варианта систем составляющих:

((Студенты (из Харькова)) (уехали (на Кавказ))).

и

(Студенты ((из Харькова) уехали)) (на Кавказ))).

Можно получить размеченные системы составляющих, добавив к скобочным структурам указания признаков синтаксических классов, удовлетворяющих данной синтаксической структуре (составляющей).

В качестве меток используются следующие символы:

S – предложение;

*V*_{xyvw} – глагол в роде *x*, числе *y*, времени *v*, лице *w*;

*VP*_{xyvw} – группа глагола в роде *x*, числе *y*, времени *v*, лице *w*;

*N*_{xyz} – имя существительное в роде *x*, числе *y*, падеже *z*;

*NP*_{xyz} – группа имени существительного в роде *x*, числе *y*, падеже *z*;

*A*_{xyz} – имя прилагательного в роде *x*, числе *y*, падеже *z*.

На рис. 6.7 показано дерево размеченных составляющих для предложения «Студентка четвертого курса перевела сложный текст».

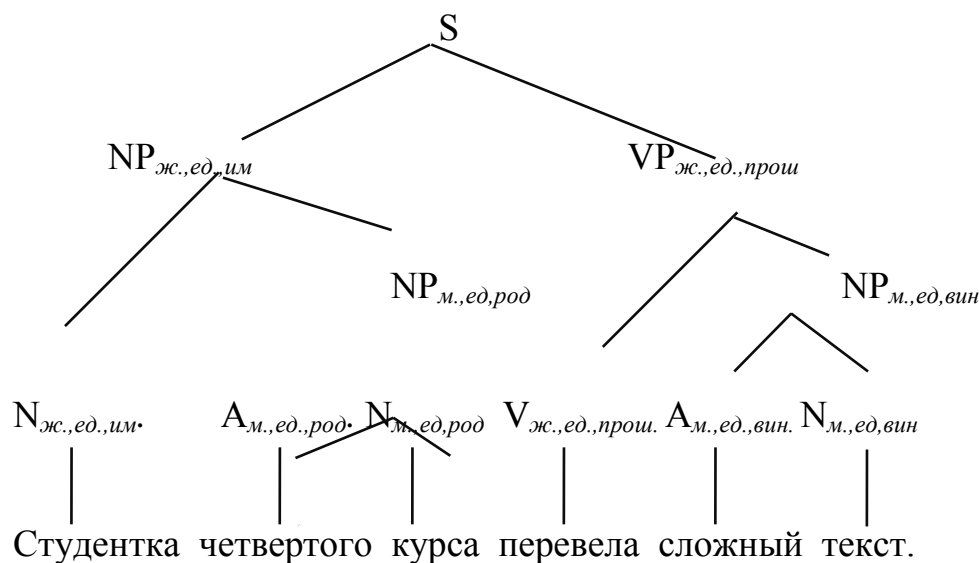


Рисунок 6.7 – Дерево размеченных составляющих

В предыдущих примерах рассматривались относительно простые предложения. Реальные предложения имеют более сложные деревья синтаксических структур. На рис. 6.8 показан пример дерева составляющих для предложения *Analyst said Mr. Stronach wants to resume a more influential role in running the company*, взятый из крупнейшего размеченного корпуса английского языка Penn Treebank⁷.

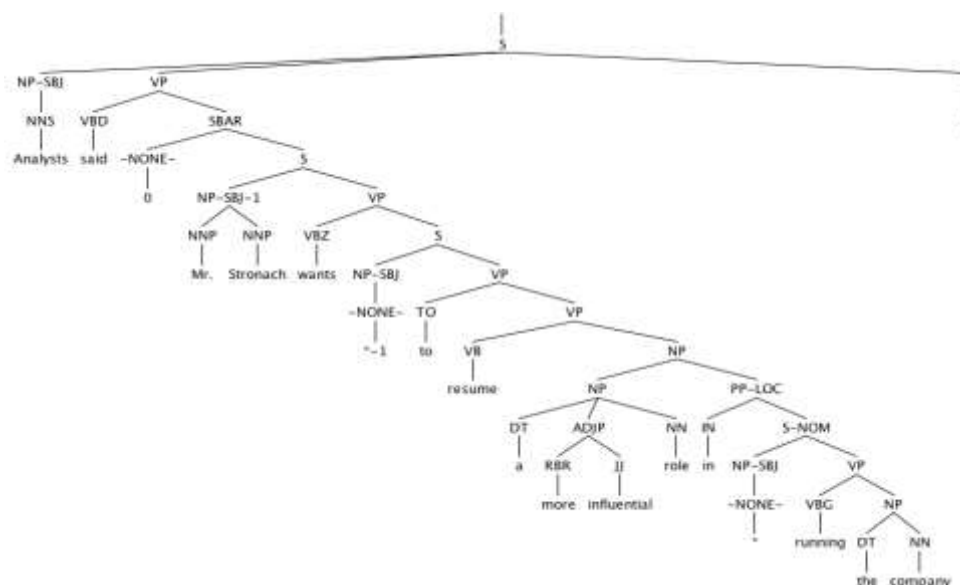


Рисунок 6.8 – Пример дерева составляющих из корпуса Penn Treebank

⁷ <https://catalog.ldc.upenn.edu/docs/LDC95T7/c193.html>

В этом примере, кроме уже рассмотренных меток POS-тегинга, выделяются следующие метки: NP (noun phrase) – группа существительного (именная группа); VP (verb phrase) – группа глагола; PP (preposition phrase) – предложная группа; - NONE- –непроизнесенные слова, называемые пустыми элементами и другие синтаксические метки.

6.5. Использование контекстно-свободных грамматик для реализации размеченных систем составляющих

Описывать системы составляющих удобно на языке контекстно-свободных грамматик (**Context-free grammars (CFGs)**). Контекстно-свободная грамматика (КСГ) представляет собой кортеж из четырех множеств: $G = (T, N, R, S)$. Это:

- Множество **терминальных** символов **T**, представляющих элементарные символы языка, определяемые грамматикой. Например, $T = \{\text{Студентка, четвертого, курса, перевела, сложный, текст}\}$.

- Множество **нетерминалов** **N**. Например, $N = \{S, NP, VP, V, NN\}$.

- Множество **продукций** **R**, каждая из которых состоит из нетерминала, называемого *заголовком* или *левой частью* продукции, стрелки и последовательности терминалов и/или нетерминалов, называемых *телом* или *правой частью* продукции. В правой части продукций контекстно-свободных грамматик может быть только один нетерминальный символ, а в левой части могут быть как терминальные, так и нетерминальные символы:

$$R \subseteq \{A \rightarrow B, \text{ где } A \in N, B \in (N \cup T)^*\}.$$

- Один из нетерминальных символов **S**, указываемый как **стартовый** или **начальный**.

Таким образом, для построения КСГ **нужно**:

- 1) выделить терминалы (слова),
- 2) определить нетерминалы, которые захватывают эквивалентные составляющие, т. е. определить продукции или правила вывода.

Например, для синтаксического разбора (parser) фразы «eat sushi with tuna» можно использовать следующую КСГ:

$G_1 = (T, N, R, S)$:

$T = \{\text{eat, sushi, with, tuna}\}$

$N = \{N, V, P, NP, VP\}$

$S = VP$

R :

$N \rightarrow \text{sushi} \mid \text{tuna}$

$V \rightarrow \text{eat}$

$P \rightarrow \text{with}$

$NP \rightarrow N \mid N\ PP$

$PP \rightarrow P\ NP \mid P\ N$

$VP \rightarrow V\ NP$

В КС-грамматиках используют графические представления вывода, называемые деревом вывода. **Вывод** представляет собой выяснение для полученной строки терминалов способа ее вывода из стартового символа грамматики. Если строка не может быть выведена из стартового символа, синтаксический анализатор сообщает об ошибке в строке. На рис. 6.9 показано дерево вывода для фразы «eat sushi with tuna».

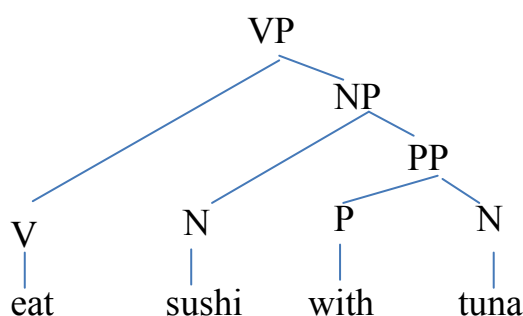


Рисунок. 6.9 – Пример дерева вывода КСГ

Контекстно свободные грамматики естественного языка являются неоднозначными. Это означает, что для одного и того же предложения может быть построено несколько деревьев разбора. Отсюда следует, что если для предложения

может быть построено несколько деревьев разбора, то это предложение является синтаксически омонимичным.

Например, если в имеющуюся у нас грамматику добыть еще одно правило и еще один нетерминальный символ, то мы сможем построить для двух подобных фраз («Eat sushi with tuna» и «Eat sushi with chopsticks») два разных, но правильных дерева разбора (рис. 6.10):

$G_1 = (T, N, R, S)$:

$T = \{\text{eat, sushi, with, tuna, chopsticks}\}$

$N = \{NN, V, P, NP, VP\}$

$S = VP$

R :

$NN \rightarrow \text{sushi} \mid \text{tuna} \mid \text{chopsticks}$

$V \rightarrow \text{eat}$

$P \rightarrow \text{with}$

$NP \rightarrow NN \mid NN PP$

$PP \rightarrow P NP \mid P NN$

$VP \rightarrow V NP \mid VP PP$

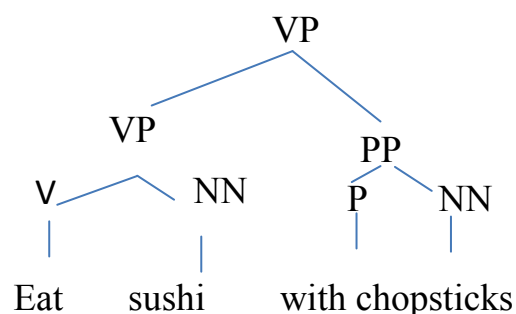
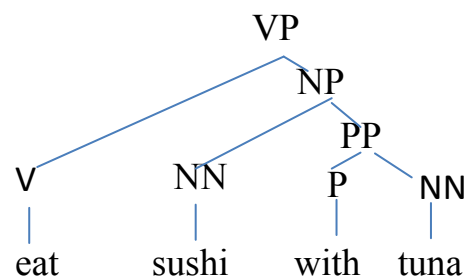


Рисунок 6.10 – Пример неоднозначности КСГ

Контрольные вопросы и практические задания по теме «Строки Python»

1. Основные способы формализации синтаксической структуры предложения.
2. Деревья синтаксической зависимости. Проективные деревья синтаксической зависимости. Размеченные деревья синтаксической зависимости.
3. Формальное определение грамматики.
4. Контекстно-свободные грамматики.
5. Синтаксический анализ. Задача синтаксического анализа.
6. Дерево разбора для КСГ.

7. Использование контекстно-свободных грамматик для реализации размеченных систем составляющих.

8. Неоднозначные контекстно-свободные грамматики и синтаксическая омонимия.

9. Показать два способа формального выражения синтаксической структуры следующих предложений:

- Дипломная работа студента включает четыре раздела.
- The small boy put the tortoise on the colored rug.

10. Показать многозначность контекстно-свободной грамматики, распознающей следующие предложения:

- Japanese eat sushi with the hands.
- Japanese eat sushi with pickled ginger.

Тема 7. СЕМАНТИЧЕСКАЯ ОБРАБОТКА. РАБОТА С ТЕЗАУРУСОМ WORDNET

7.1. Общее описание лингвистического ресурса WordNet.

7.2. Работа со словарями WordNet с помощью NLTK.

7.3. Иерархия семантических отношений WordNet.

7.4. Использование WordNet для определения семантической близости слов.

Контрольные вопросы и практические задания по теме «Семантическая обработка. Работа с тезаурусом WordNet».

7.1. Общее описание лингвистического ресурса WordNet.

Лингвистический ресурс или тезаурус WordNet разработан в Принстонском университете США. **WordNet** относится к классу лексических онтологий и свободно доступен в Интернете по ссылке <http://wordnetweb.princeton.edu/perl/webwn>.

Разработка тезауруса была начата в 1984 году и первоначально он создавался как модель человеческой памяти. В 1995 году WordNet появился в Интернете в свободном доступе.

К настоящему времени WordNet представляет собой семантически ориентированный словарь английского языка, подобный традиционному тезаурусу, но отличающийся более сложной структурой; он охватывает приблизительно 155 тысяч различных лексем и словосочетаний, организованных в 117 тысяч понятий, или совокупностей синонимов (**synset**); общее число пар "лексема-значение" насчитывает 200 тысяч.

В состав словаря входят лексемы, относящиеся к четырем частям речи: прилагательному, существительному, глаголу и наречию. Лексемы различных частей речи хранятся отдельно, и описания, соответствующие каждой части речи, имеют различную структуру. Основным отношением в WordNet является отношение синонимии. Наборы синонимов (синсеты) – основные структурные элементы WordNet. **Синсет** может рассматриваться как представление лексикализованного понятия (концепта) английского языка; оно обозначает абстрактное понятие, ко-

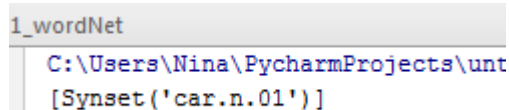
торое не всегда соответствует определенному слову. Большинство синсетов снабжены толкованием, подобным толкованиям в традиционных словарях; это толкование рассматривается как одно для всех синонимов синсета. Если слово имеет несколько значений, то оно входит в несколько различных синсетов.

7.2. Работа со словарями WordNet с помощью NLTK

Для работы со словарями WordNet необходимо импортировать одноименный модуль пакета NLTK. NLTK включает английский WordNet, содержащий 155 287 слов и 117 659 синсетов. WordNet включает четыре словаря: N (nouns), V (verbs), ADJ (adjectives) и ADV (adverbs).

Следующий фрагмент программного кода позволяет получить синсеты слова motorcar:

```
from nltk.corpus import wordnet as wn
print(wn.synsets('motorcar'))
```



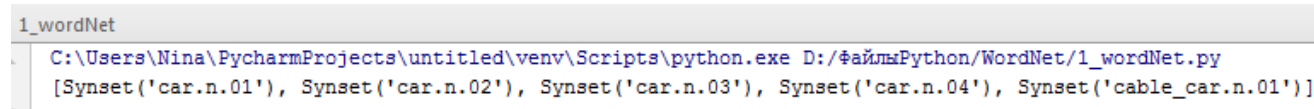
```
1_wordNet
C:\Users\Nina\PycharmProjects\untitled
[Synset('car.n.01')]
```

Таким образом, слово motorcar имеет только одно значение (один синсет), и это значение соответствует первому значению существительного car.

В результате применения того же кода к слову car, а именно

```
from nltk.corpus import wordnet as wn
print(wn.synsets('car'))
```

можно увидеть, что у данного слова 5 синонимичных множеств (синсетов) или лемм



```
1_wordNet
C:\Users\Nina\PycharmProjects\untitled\venv\Scripts\python.exe D:/файлыPython/WordNet/1_wordNet.py
[Synset('car.n.01'), Synset('car.n.02'), Synset('car.n.03'), Synset('car.n.04'), Synset('cable_car.n.01')]
```

Визуализация поиска данного синсета на сайте WordNet⁸ показана на рис. 7.1.

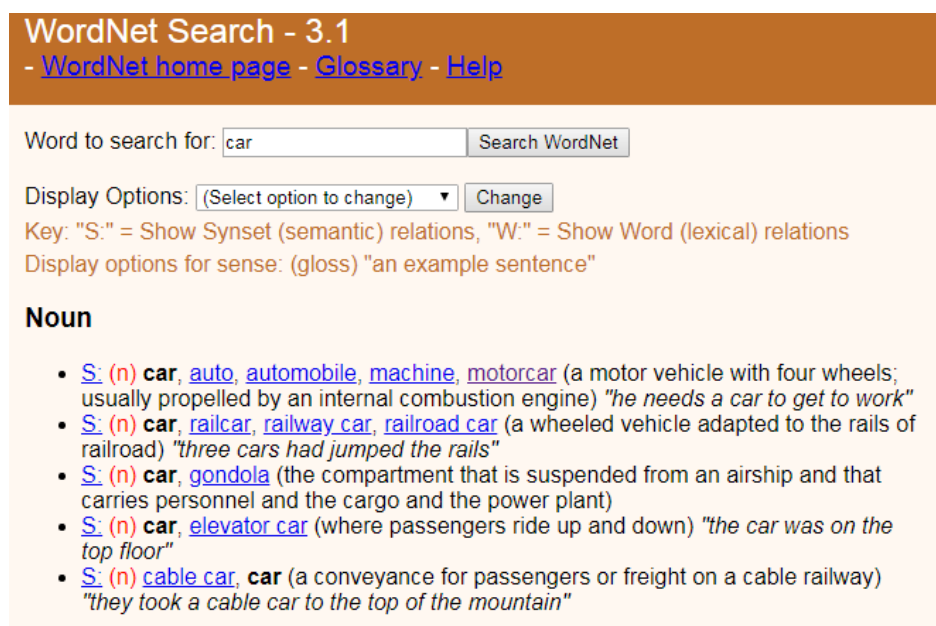


Рисунок 7.1 – Поиск синсетов слова car на сайте WordNet

Получить список лемм, относящихся к определенному синсету, можно с помощью функции **lemma_names()**, а получить определение леммы (слова в конкретном его значении) и пример ее применения можно с помощью функций **definition()** и **examples()** соответственно:

```
from nltk.corpus import wordnet as wn
print(wn.synsets('car'))
print(wn.synset('car.n.02').lemma_names())
print()
for synset in wn.synsets('car'):
    print(synset.lemma_names())
print()
print(wn.synset('car.n.02').definition())
print(wn.synset('car.n.02').examples())
```

⁸ <http://wordnetweb.princeton.edu/perl/webwn>


```

1_wordNet
[Synset('car.n.01'), Synset('car.n.02'), Synset('car.n.03'), Synset('car.n.04'), Synset('cable_car.n.01')]
['car', 'railcar', 'railway_car', 'railroad_car']

['car', 'auto', 'automobile', 'machine', 'motorcar']
['car', 'railcar', 'railway_car', 'railroad_car']
['car', 'gondola']
['car', 'elevator_car']
['cable_car', 'car']

a wheeled vehicle adapted to the rails of railroad
['three cars had jumped the rails']

```

Можно получить имена, определения и примеры синсетов только определенной части речи слова:

```

print('имена и определения синсетов существительных')
print()
for synset in wn.synsets('work',wn.NOUN):
    print(synset.name() + ' определение : ' + synset.definition())
print()
print('имена и определения синсетов глаголов')
print()
for synset in wn.synsets('work',wn.VERB):
    print(synset.name() + ' определение : ' + synset.definition())
print()

```

имена и определения синсетов существительных

```

work.n.01 определение : activity directed toward making or doing something
work.n.02 определение : a product produced or accomplished through the effort or activity or age
employment.n.02 определение : the occupation for which you are paid
study.n.02 определение : applying the mind to learning and understanding a subject (especially k
work.n.05 определение : (physics) a manifestation of energy; the transfer of energy from one phy
workplace.n.01 определение : a place where work is done
oeuvre.n.01 определение : the total output of a writer or artist (or a substantial part of it)

```

имена и определения синсетов глаголов

```

work.v.01 определение : exert oneself by doing mental or physical work for a purpose or out of r
work.v.02 определение : be employed
work.v.03 определение : have an effect or outcome; often the one desired or expected
function.v.01 определение : perform as expected when applied
work.v.05 определение : shape, form, or improve a material
exercise.v.03 определение : give a workout to

```

Для снятия многозначности слово идентифицируется как `car.n.01.automobile`, `car.n.01.motorcar` и так далее. Такая совместная запись слова и синсета и представляет собой **лемму**.

Для того чтобы получить все леммы заданного синсета, нужно использовать функцию **lemmas()** для заданного синсета:

```
print(wn.synset('car.n.02').lemmas())
```

```
1_wordNet
[Lemma('car.n.02.car'), Lemma('car.n.02.railcar'), Lemma('car.n.02.railway_car'), Lemma('car.n.02.railroad_car')]
```

Можно получить все леммы слова с помощью функции **lemmas('word')** модуля `wordnet`, параметром которой будет слово:

```
print(wn.lemmas('car'))
```

```
1_wordNet
[Lemma('car.n.01.car'), Lemma('car.n.02.car'), Lemma('car.n.03.car'), Lemma('car.n.04.car'), Lemma('cable_car.n.01.car')]
```

Для того чтобы, наоборот, получить синсет по заданной лемме, используется функция **synset()**:

```
print(wn.synset('car.n.02').lemmas())
print(wn.lemma('car.n.02.railcar').synset())
print(wn.lemma('car.n.02.railcar').name())
```

```
1_wordNet
[Lemma('car.n.02.car'), Lemma('car.n.02.railcar'), Lemma('car.n.02.railway_car'), Lemma('car.n.02.railroad_car')]
Synset('car.n.02')
railcar
```

7.3. Иерархия семантических отношений WordNet

Кроме отношений синонимии между существительными в словаре установлены следующие семантические отношения:

- **антонимия**;

– **гипонимия/гиперонимия** – отношение, которое иначе может быть названо ВЫШЕ-НИЖЕ, РОД-ВИД или **is A – отношение**. Это отношение является центральным отношением для описания существительных, при этом видовой синсет называется **гипонимом**, а родовой – **гиперонимом**. Отношение транзитивно и несимметрично. Гипоним наследует все свойства гиперонима. Синсет *X* называется гипонимом синсета *Y*, если носители английского языка считают нормальными предложения типа " *An X is a (kind of) Y* ". Такие отношения гипонимии и гиперонимии еще называют **лексическими отношениями**;

– **меронимия** (отношение ЧАСТЬ-ЦЕЛОЕ). Внутри этого отношения выделяются отношения **быть_элементом** и **быть_сделанным_из**.

Таким образом, отношения между синсетами образуют иерархическую структуру. WordNet разделяет существительные на несколько иерархий, каждая из которых имеет свое начальное понятие, называемое корнем синсета (**root synsets** или **unique beginner**). Всего для существительных имеется 25 начальных понятий или синсетов верхнего уровня, таких, как {entity}, {state}, {event}, {act, activity}, {animal, fauna}, {artifact}, {food}, {process}, {quantity, amount} и др.

Используя интерфейс WordNet, можно получить фрагмент ее гиперонимической (гипонимической) иерархии.

Например, для синсета stream, watercourse (a natural body of running water flowing on or under the earth) фрагмент иерархической сети будет иметь вид, представленный на рис. 7.2.

На данном графе узлы соответствуют синсетам, а дуги показывают гиперонимические или гипонимические отношения, т. е. отношения между вышестоящими и подчиненными концептами.

Модуль wordnet пакета nltk.corpus позволяет легко осуществлять иерархические перемещения между концептами. Для того чтобы получить нижестоящие концепты данного синсета (более специфические понятия) или их гипонимы, используется метод **hyponyms()**. А для того чтобы получить вышестоящий концепт данного синсета (ближайшее более общее понятие) или его гипероним, использу-

ется метод **hypernyms()**. Для того чтобы получить корневой гипероним (root hypernyms) данного понятия, используется метод **root_hypernyms()**:

```
from nltk.corpus import wordnet as wn  
stream=wn.synset('stream.n.01')  
print(stream.lemmas())  
types_of_stream=stream.hypernyms()  
print(types_of_stream)  
print (stream.hypernyms())  
print(stream.root_hypernyms())
```

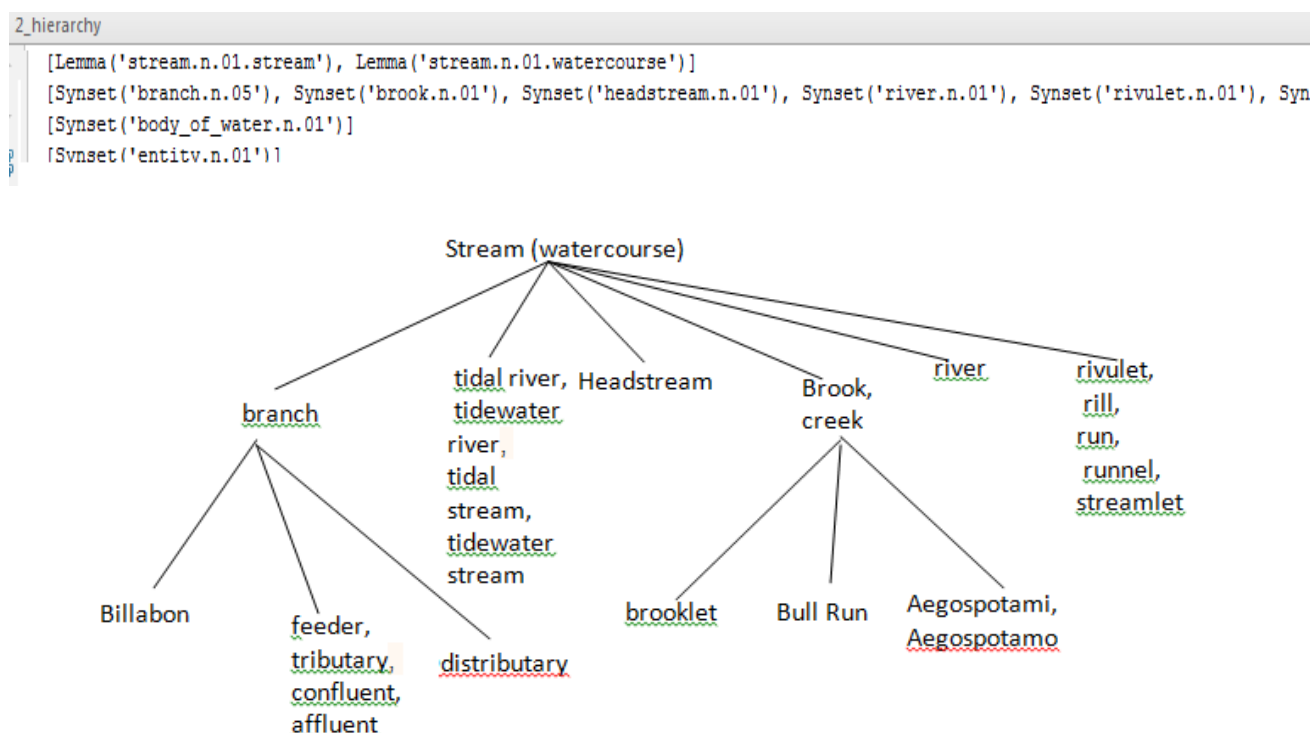


Рисунок 7.2 – Фрагмент иерархии отношений для для синсета stream, watercourse (a natural body of running water flowing on or under the earth) в WordNet

Кроме отношений гипонимии и гиперонимии в WordNet существуют отношения меронимии (**meronyms**) и холонимии (**holonyms**). Отношение меронимии – это отношение между предметом и его составляющими, а отношение холонимии – это противоположное отношение между частью и целым. Например, частями stream (поток) являются ford (брод), meander (извилина, изгиб) и midstream (середина потока). Связь между понятием stream и

понятиями `meander`, `ford` и `midstream` устанавливается с помощью функции **`part_meronyms()`**:

```
from nltk.corpus import wordnet as wn
stream=wn.synset('stream.n.01')
print (stream.part_meronyms())
```

Эти же отношения понятия `stream` можно увидеть по запросу на сайте WordNet (рис. 7.3).

Word to search for:

Display Options:

Key: "S:" = Show Synset (semantic) relations, "W:" = Show Word (lexical) relations
Display options for sense: (gloss) "an example sentence"

Noun

- **S: (n) stream, watercourse** (a natural body of running water flowing on or under the earth)
 - [direct hyponym](#) / [full hyponym](#)
 - [part meronym](#)
 - **S: (n) ford, crossing** (a shallow area in a stream that can be forded)
 - **S: (n) meander** (a bend or curve, as in a stream or river)
 - **S: (n) midstream** (the middle of a stream)
 - [direct hypernym](#) / [inherited hypernym](#) / [sister term](#)
 - [derivationally related form](#)
- **S: (n) stream, flow, current** (dominant course (suggestive of running water) of

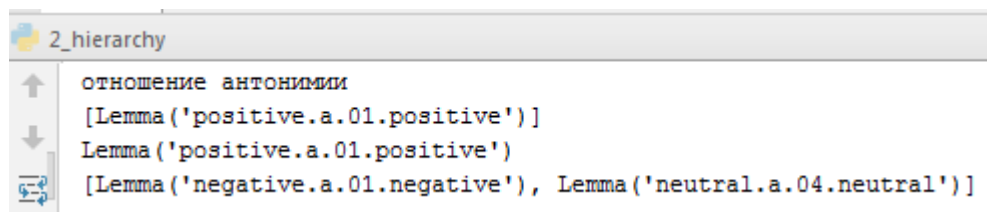
Рисунок 7.3 – Отношение меронимии для синсета `stream, watercourse` (a natural body of running water flowing on or under the earth) в WordNet

Метод **`substance_meronyms()`** показывает вещество, из которого состоит данное понятие. Например, `tree` включает в себя `heartwood` и `sapwood`. Обратное отношение определяется методом **`substance_holonyms()`**. Метод **`member_holonyms()`** показывает отношение включения понятия в некую общность. Например, коллекция `trees` представляет собой `forest`.

Для получения понятий, значение которых противоположно данному синсету, используется метод **`antonymy()`**:

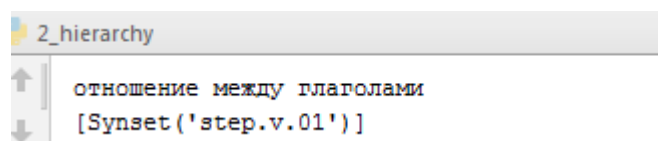
```
print('отношение антонимии')
print(wn.synset('positive.a.01').lemmas())
```

```
print(wn.lemma('positive.a.01.positive'))
print(wn.lemma('positive.a.01.positive').antonyms())
```



В предыдущем примере было рассмотрено отношение антонимии для прилагательных. Кроме вышеописанных, в WordNet существуют также специальные отношения, устанавливаемые между глаголами. Метод **entailment()** показывает отношение включения (вовлечения) одного действия в другое. Например, walking влечет за собой stepping:

```
print('отношение между глаголами')
print(wn.synset('walk.v.01').entailments())
```



На рис. 7.4 показана визуализация данного отношения для глагола walk WordNet.



Рисунок 7.4 – Визуализация отношения «вовлечения в действие» для глагола walk на сайте WordNet

Обратите внимание, что не все слова и понятия (синсеты), представленные в WordNet, обладают всем комплексом имеющихся отношений.

7.4. Использование WordNet для определения семантической близости слов

В WordNet синсеты связаны сложной сетью лексических отношений. По заданному конкретному синсету можно перемещаться по сети WordNet для того, чтобы найти синсет со связанным значением.

Каждый синсет имеет один или больше путей гиперонимов, которые связывают его с корневым гиперонимом – `entity.n.01`. Два синсета, связанные с одним и тем же корневым гиперонимом, могут иметь несколько общих гиперонимов на более низких уровнях. Если два синсета совместно относятся к одному, низко стоящему в иерархии гиперониму (более специальному, а не общему понятию), то они могут иметь очень близкое семантическое значение. Метод **`lowest_common_hyponyms()`** для заданного синсета позволяет определить наименьший общий гипероним для данного синсета и синсета, заданного в качестве параметра.

Например, согласно рис. 7.1 наименьшим общим гиперонимом для слова `billabong` и слова `feeder` является слово `branch`. Однако, очевидно, что отношения семантической близости устанавливается не между словами, а между понятиями или синсетами:

```
from nltk.corpus import wordnet as wn
print(wn.synsets('branch'))
print(wn.synset('branch.n.05').hyponyms())
print(wn.synset('billabong.n.02').lowest_common_hyponyms(wn.synset('feeder.n.03'))))
print(wn.synset('billabong.n.02').lowest_common_hyponyms(wn.synset('work.n.01'))))
```

В результате данного программного кода видно, что наименьшим общим гиперонимом у синсета `billabong.n.02` и синсета `feeder.n.03` является синсет `'branch.n.05'`. В то время как у совсем разных понятий `billabong.n.02` и `work.n.01` наименьшим общим гиперонимом является корневой гипероним `'entity.n.01'`:


```
[Synset('branch.n.01'), Synset('branch.n.02'), Synset('branch.n.03'), Synset('outgrowth.n.01'), Synset('branch.n.05'),
[Synset('billabong.n.02'), Synset('tributary.n.01'), Synset('feeder.n.03')]
[Synset('branch.n.05')]
[Synset('entity.n.01')]
```

Метод **min_depth()** позволяет определить минимальное расстояние в иерархической сети от данного синсета до корневого синсета 'entity.n.01'. Например, минимальное расстояние (или минимальная глубина) синсета 'entity.n.01' равна 0, минимальная глубина до синсета billabong.n.02 равна 6, а минимальная глубина синсета 'branch.n.05' равна 5:

```
print('минимальная глубина')
print('минимальная глубина синсета entity.n.01 равна',
wn.synset('entity.n.01').min_depth())
print('минимальная глубина синсета billabong.n.02 равна',
wn.synset('billabong.n.02').min_depth())
print('минимальная глубина синсета branch.n.05 равна',
wn.synset('branch.n.05').min_depth())
```

```
Semantic similarity
минимальная глубина
минимальная глубина синсета entity.n.01 равна 0
минимальная глубина синсета billabong.n.02 равна 6
минимальная глубина синсета branch.n.05 равна 5
```

Меры семантического сходства, которые определяются по словарю WordNet, базируются на понятии минимальной глубины. Например, метод **path_similarity()** определяет значение в диапазоне от 0 до 1 и семантическую близость как величину, обратную длине самого короткого пути между двумя концептами в иерархии гиперонимов WordNet. Метод возвращает -1 в том случае, если пути между двумя концептами не существует. Пусть длина пути от концепта к самому себе равна 1:

```
print('семантическая близость')
print('семантическая близость branch.n.05 и billabong.n.02 равна',\
wn.synset('branch.n.05').path_similarity(wn.synset('billabong.n.02'))
print('семантическая близость branch.n.05 и branch.n.05 равна',\
```



```
wn.synset('branch.n.05').path_similarity(wn.synset('branch.n.05'))  
print('семантическая близость branch.n.05 и entity.n.01 равна',\  
wn.synset('branch.n.05').path_similarity(wn.synset('entity.n.01')))
```

```
Semantic similarity  
семантическая близость  
семантическая близость branch.n.05 и billabong.n.02 равна 0.5  
семантическая близость branch.n.05 и branch.n.05 равна 1.0  
семантическая близость branch.n.05 и entity.n.01 равна 0.16666666666666666
```

Контрольные вопросы и практические задания по теме «Семантическая обработка. Работа с тезаурусом WordNet»

1. Получите все синсеты слова “dish”. Зайдите на сайт WordNet и убедитесь в правильности полученных синсетов. В отчете приведите скриншот результата поиска данного слова в WordNet.
2. Получите определение и пример использования леммы третьего синсета существительного и первого синсета глагола слова “dish”.
3. Получите все имена, определения и примеры использования синсетов существительного “dish”.
4. Получите все имена, определения и примеры использования синсетов глагола “dish”.
5. Получите лемму и имя данной леммы любого синсета слова “dish”.
6. Получите все нижестоящие и вышестоящие в родо-видовой иерархии WordNet концепты первого синсета существительного слова “work”. Получите корневой гипероним данного концепта. Объясните запись полученных концептов. Приведите скриншот данной иерархии с сайта словаря WordNet.
7. С помощью WordNet определите, из каких веществ состоит “wood” (используйте первую лемму существительного данного слова) и частью каких веществ оно является. Приведите скриншот с сайта словаря WordNet, подтверждающий правильность полученного результата.
8. Получите антоним понятия “horizontal”. Приведите скриншот с сайта словаря WordNet, подтверждающий правильность полученного результата.
9. Покажите, какие действия вовлечены в понятие “eat”.

10. Найдите наименьший общий гипоним синсетов 'bowl.n.02' и 'polyhedron.n.01'.

11. Найдите наименьший общий гипоним синсетов 'bowl.n.01' и 'polyhedron.n.01'.

12. С помощью скриншотов с сайта словаря WordNet объясните разницу результатов 10-го и 11-го пунктов задания.

13. Определите семантическое расстояние между синсетами job.n.07 и work.n.01.

14. Определите семантическое расстояние между синсетами job.n.07 и job.n.07.

15. Определите семантическое расстояние между синсетами job.n.07 и entity.n.01

16. С помощью скриншотов с сайта словаря WordNet объясните разницу результатов 10-го и 11-го пунктов задания.

Тема 8. РАБОТА С КОРПУСАМИ В NLTK

8.1. Работа с модулем `nltk.corpus` пакета NLTK.

8.1.1. Корпус Проекта Гутенберга.

8.1.2. Корпус `webtext`.

8.1.3. Корпус `nps_chat`.

8.1.4. Корпус Брауна (`Brown corpus`).

8.1.5. Reuters-корпус.

8.1.6. Корпус `inaugural`.

8.2. Текстовые корпуса на различных языках. Использование кодировок.

8.3. Загрузка собственного корпуса.

8.4. Работа с размеченными корпусами в NLTK.

Контрольные вопросы и практические задания по теме «Работа с корпусами в `nltk`».

8.1. Работа с модулем `nltk.corpus` пакета NLTK

Модуль **`nltk.corpus`** пакета **`nltk`** предназначен для доступа к корпусам и другим лексическим ресурсам. В широком понимании «текстовый корпус» представляет собой большой массив собранных по определенному принципу (жанру, тематике, времени и т. д.) текстов. Обычно корпус содержит множество текстовых файлов, которые при анализе корпуса рассматриваются как единый объект (один текст).

Самые простые текстовые корпуса не имеют никакой структуры и представляют собой простую коллекцию текстов (например, `Gutenberg corpus`, `Webtext corpus`). Однако чаще всего текстовые корпуса представлены определенными структурами. Обычно тексты в таких корпусах сгруппированы по категориям, которые могут соответствовать жанрам, источникам, авторам, языкам и т. д. (например, `Brown corpus`). Иногда такие категории пересекаются, особенно в случае тематических категорий, когда текст может быть релевантен одновременно нескольким темам (например, `Reuter corpus`). Текстовые коллекции могут также иметь временную структуру, наиболее общим примером которой, являются коллекции новостей (например, `Inaugural corpus`).

Пакет NLTK поддерживает эффективный доступ к различным корпусам и может быть использован для работы с новыми корпусами. Основные функции модуля **nltk.corpus**:

- **fileids** () – возвращает список файлов данного корпуса;
- **fileids** ([**categories**]) – возвращает список файлов данного корпуса, которые относятся к названным категориям;
- **categories** () – возвращает категории корпуса;
- **categories** ([**fileids**]) – возвращает категории корпуса, которым соответствуют файлы списка;
- **raw**() – возвращает необработанный контент корпуса, без какой-либо лингвистической обработки;
- **raw** (**fileids**=[**f1**, **f2**, **f3**]) – возвращает необработанный контент выбранных файлов;
- **raw** (**categories** =[**c1**, **c2**]) – возвращает необработанный контент выбранных категорий;
- **words** () – возвращает список слов полного корпуса;
- **words** (**fileids**=[**f1**, **f2**, **f3**]) – возвращает список слов выбранного файла;
- **words** (**categories** =[**c1**, **c2**]) – возвращает список слов списка заданных категорий;
- **sents** () – делит полный корпус на предложения и возвращает список списков слов каждого предложения текста;
- **sents** (**fileids**=[**f1**, **f2**, **f3**]) – делит текст выбранных файлов на предложения и возвращает список списков слов каждого предложения текста;
- **sents** (**categories** =[**c1**, **c2**]) – делит текст выбранных категорий на предложения и возвращает список списков слов каждого предложения текста;
- **abspath**(**fileid**) – определяет абсолютный путь к файлу на диске;
- **encoding**(**fileid**) – выполняет кодирование файла;
- **open**(**fileid**) – открывает заданный файл корпуса для чтения;

- **root()** – возвращает путь к корневому каталогу установленного на компьютер корпуса;
- **readme()** – возвращает содержание файла **README** корпуса, если он есть.

8.1.1. Корпус Проекта Гутенберга

Библиотека **Project Gutenberg** <http://www.gutenberg.org/catalog/> содержит более 5700 электронных книг художественной литературы свободного доступа в форматах ASCII text, epub books и kindle books, которые можно скачать или читать online. В основном здесь размещены книги на английском языке, но есть также тексты на других языках (Catalan, Chinese, Dutch, Finnish, German, Italian, Portuguese и Spanish).

Для того чтобы скачать текст с данного сайта, можно воспользоваться пакетом **requests**. Так как библиотека **requests** представляет собой надстройку над низкоуровневой библиотекой **urllib3**, то пакет **urllib3** должен быть так же предварительно установлен (рис. 8.1).

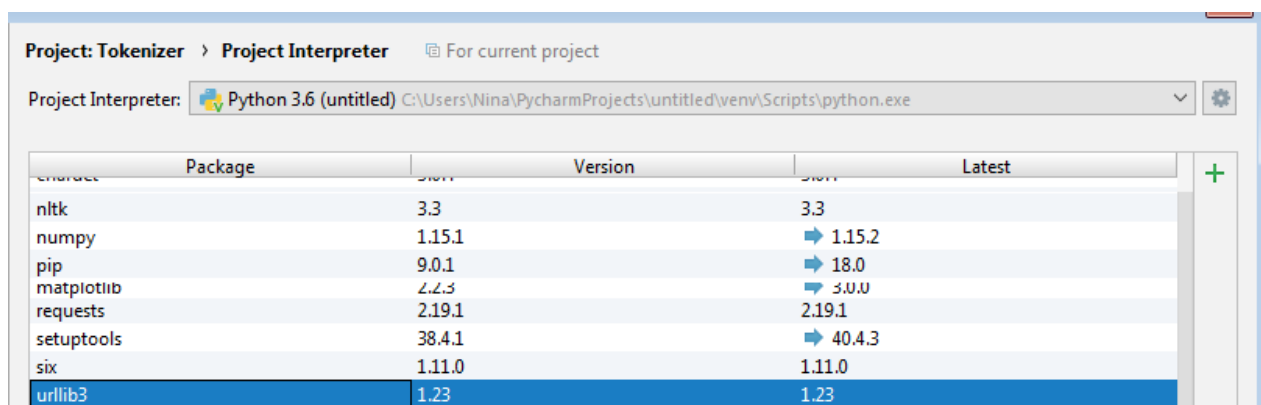


Рисунок 8.1 – Установка пакета **requests** в PyCharm

Python-библиотека **requests** предназначена для выполнения запросов к Веб-серверу и обработки ответов. Пользуясь данной библиотекой, можно получить содержимое веб-страницы в виде html для дальнейшей обработки.

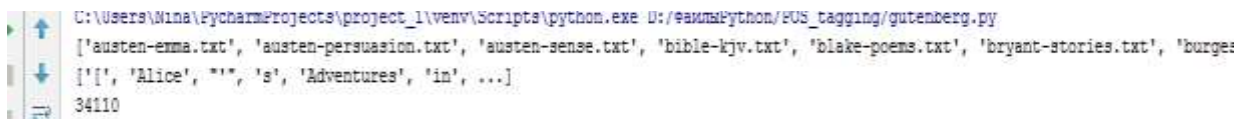
Например, для того чтобы скачать CRIME AND PUNISHMENT By Fyodor Dostoevsky в файл text.txt, можно воспользоваться следующим фрагментом программного кода:

```
import requests
f=open(r'text.txt','wb')
ufr=requests.get ("http://www.gutenberg.org/files/2554/2554-0.txt")
f.write (ufr.content)
f.close()
```

Модуль **nltk.corpus.gutenberg** предназначен для работы с корпусом библиотеки **gutenberg**.

Например, используя функцию **fileids()**, можно получить информацию об именах файлов, содержащихся в корпусе Гуттенберга, а затем, используя функцию **words(fileids)**, получить список слов выбранного файла и определить количество слов в данном списке:

```
import nltk
print(nltk.corpus.gutenberg.fileids())
carroll=nltk.corpus.gutenberg.words("carroll-alice.txt")
print (carroll)
print(len(carroll)) # количество слов в тексте произведения "Алиса в стране чудес"
```



Используя цикл **for** можно получить некоторые статистические характеристики каждого файла корпуса текстов проекта Гуттенберга:

- среднюю длину слова в корпусе, равную количеству символов в корпусе, деленному на количество слов;
- среднее количество слов в предложении корпуса, равное количеству слов, деленному на количество предложений;
- размер глоссария корпуса, т. е. количество уникальных токенов корпуса;
- оценку лексического разнообразия — количество появлений каждого уникального токена в корпусе:

```

from nltk.corpus import gutenberg
for files in gutenberg.fileids():
    num_chars=len(gutenberg.raw(files)) # количество букв в тексте
    num_words=len(gutenberg.words(files))
    num_sents=len(gutenberg.sents(files))
    print(files, "average word length =", int(num_chars/num_words), \
          "average sentence lenght =", int(num_words/num_sents))
    num_vocab=len(set(w.lower() for w in gutenberg.words(files)))
    print ("Уникальных токенов= ", num_vocab, " лексическое разнообразие = ", \
          int(num_words/num_vocab))

```

```

gutenberg
austen-emma.txt average word length = 4 average sentence lenght = 24
Уникальных токенов= 7344 лексическое разнообразие = 26
austen-persuasion.txt average word length = 4 average sentence lenght = 26
Уникальных токенов= 5835 лексическое разнообразие = 16
austen-sense.txt average word length = 4 average sentence lenght = 28
Уникальных токенов= 6403 лексическое разнообразие = 22
bible-kjv.txt average word length = 4 average sentence lenght = 33

```

Используя функцию **sents (fileids)**, можно разделить текст на предложения и получить список списков слов каждого предложения:

```

print(gutenberg.sents("shakespeare-hamlet.txt "))
print(gutenberg.sents("shakespeare-hamlet.txt ")[23])

```

```

gutenberg
[[['', 'The', 'Tragedie', 'of', 'Hamlet', 'by', 'William', 'Shakespeare', '1599', ''], ['Actus', 'Primus', '.'], ...]
['Not', 'a', 'Mouse', 'stirring']

```

8.1.2. Корпус webtext

Корпус **webtext** содержит менее формальный язык, чем корпус Гутенберга. Он включают тексты форума «*Firefox discussion forum*», сценарий фильма *Pirates of the Caribbean*, личные рекламные объявления, рецензии вин и др.

Ниже приведен фрагмент кода, позволяющий увидеть имена файлов корпуса, количество слов в каждом файле и первые 50 символов каждого файла:

```

from nltk.corpus import webtext

for files in webtext.fileids():

    print (files, len( webtext.words(files)), webtext.raw(files)[:50], '...')

```

```

web_texts
firefox.txt 102457 Cookie Manager: "Don't allow sites that set remove ...
grail.txt 16967 SCENE 1: [wind] [clop clop clop]
KING ARTHUR: Who ...
overheard.txt 218413 White guy: So, do you have any plans for this even ...
pirates.txt 22679 PIRATES OF THE CARRIBEAN: DEAD MAN'S CHEST, by Ted ...
singles.txt 4867 25 SEXY MALE, seeks attrac older single lady, for ...
wine.txt 31350 Lovely delicate, fragrant Rhone wine. Polished lea ...

```

8.1.3. Корпус nps_chat

Корпус мгновенных сообщений чата «Naval Postgraduate School» собран для исследований по автоматическому определению интернет-злоумышленников. Данный корпус содержит более чем 10 000 постов, имена авторов которых заменены на “UserNNN”. Корпус включает 15 файлов, каждый из которых содержит сотни постов, сгруппированных по дате для каждой возрастной группы (тинэйджеры, 20-летние, 30-летние, 40-летние, chatroom для взрослых). Имя файла содержит дату, имя чата и номер поста. Например, файл 10-19-20s_706post.xml включает 706 постов, собранных из chatroom 20-летних 19.10.2006:

```

from nltk.corpus import nps_chat

for files in nps_chat.fileids():

    print (files)

chatroom=nps_chat.posts('10-19-40s_686posts.xml')

print (chatroom)

for chat in chatroom:

    print (chat)

```

```

10-19-20s_706posts.xml
10-19-30s_705posts.xml
10-19-40s_686posts.xml
10-19-adults_706posts.xml
10-24-40s_706posts.xml
10-26-teens_706posts.xml
11-06-adults_706posts.xml
11-08-20s_705posts.xml
11-08-40s_706posts.xml

```



```

11-09-20s_706posts.xml
11-09-40s_706posts.xml
11-09-adults_706posts.xml
11-09-teens_706posts.xml
[['hi', 'U23'], ['love', 'me', 'like', 'a', 'bomb', 'baby', 'com
['hi', 'U23']
['love', 'me', 'like', 'a', 'bomb', 'baby', 'come', 'on', 'get',
['hi', 'U23']
['Pour', 'some', 'sugar', 'on', 'me', 'babe', 'n', 'get', 'see',
['JOIN']

```

8.1.4. Корпус Брауна (Brown corpus)

Модуль `nltk.corpora` библиотеки `nltk` также включает Корпус Брауна (Brown corpus), представляющий первый миллионный электронный корпус текстов английского языка, созданный в 1961 году в Университете Брауна. Корпус содержит тексты из 500 источников, категоризованных по жанрам (новости, статьи, художественная литература, юмор и т. д.).

Можно получить список категорий корпуса и список слов конкретной категории корпуса Брауна:

```

from nltk.corpus import brown
print("Категории корпуса Браун")
print(brown.categories())
print("Список слов, категории hobbies ")
print(brown.words(categories='hobbies'))
print("Список файлов, категории hobbies ")
print(brown.fileids(categories='hobbies'))
print("Список слов, файла 'ce03' ")
print(brown.words(fileids=['ce03']))

```

```

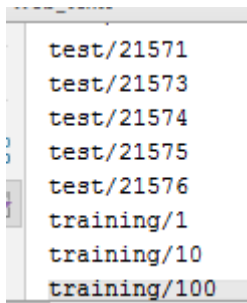
Категории корпуса Браун
['adventure', 'belles_lettres', 'editorial', 'fiction', 'government', 'hobbies', 'humor', 'learned',
Список слов, категории hobbies
['Too', 'often', 'a', 'beginning', 'bodybuilder', ...]
Список файлов, категории hobbies
['ce01', 'ce02', 'ce03', 'ce04', 'ce05', 'ce06', 'ce07', 'ce08', 'ce09', 'ce10', 'ce11', 'ce12', 'ce
Список слов, файла 'ce03'
['Five', ',', 'four', ',', 'three', ',', 'two', ',', ...]

```

8.1.5. Reuters-корпус

Reuters-корпус включает 10 788 документов новостей, всего содержащих 1,3 миллионов слов. Документы классифицированы по 90 темам и группированы в два множества: “train” и “test”. Например, файл “test/21576” представляет собой документ из тестируемого множества:

```
for files in reuters.fileids():  
    print (files)
```

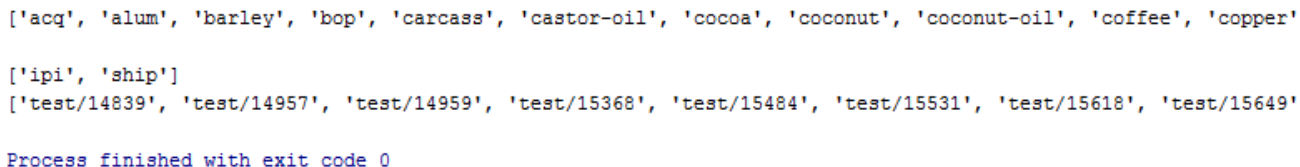


```
test/21571  
test/21573  
test/21574  
test/21575  
test/21576  
training/1  
training/10  
training/100
```

В отличие от корпуса Брауна, категории в Reuters-корпусе пересекаются между собой, т. е. одна и та же новость может одновременно принадлежать нескольким темам.

С данным корпусами также могут использоваться функции модуля `nltk.corpora`:

```
from nltk.corpus import reuters  
print(reuters.categories())  
print()  
print(reuters.categories(['test/21575','test/21574']))  
print(reuters.fileids(['barley','alum','ship']))
```



```
-----  
['acq', 'alum', 'barley', 'bop', 'carcass', 'castor-oil', 'cocoa', 'coconut', 'coconut-oil', 'coffee', 'copper'  
['ipi', 'ship']  
['test/14839', 'test/14957', 'test/14959', 'test/15368', 'test/15484', 'test/15531', 'test/15618', 'test/15649'  
Process finished with exit code 0
```

8.1.6. Корпус inaugural

Еще один корпус, который включает библиотека корпусов `nltk`, содержит 55 текстов иннаугурационных выступлений президентов США. Имя каждого файла

содержит год инаугурации:

```
from nltk.corpus import inaugural
print("Inaugural Corpus")
print(inaugural.fileids())
print ([fileid[:4]for fileid in inaugural.fileids()])
```

```
Inaugural Corpus
['1789-Washington.txt', '1793-Washington.txt', '1797-Adams.txt', '1801-Jefferson.txt', '1805-Jef
['1789', '1793', '1797', '1801', '1805', '1809', '1813', '1817', '1821', '1825', '1829', '1833',
```

8.2. Текстовые корпуса на различных языках. Использование кодировок

Кроме вышеперечисленных корпусов, библиотека NLTK содержит много текстовых корпусов различных языков. Корпус **udhr** пакета **nltk.corpus** представляет собой Всеобщую декларацию по правам человека (**Universal Declaration of Human Rights**), представленную более чем на 300 языках. Имя каждого включенного в **udhr** корпус файла содержит название языка и используемую в файле кодировку. Например, **UTF8**, **Latin1** или **Cyrillic**:

```
from nltk.corpus import udhr
print(udhr.fileids())
```

```
['Abkhaz-Cyrillic+Abkh', 'Abkhaz-UTF8', 'Achehnese-Latin1', 'Achuar-Shiwiar-Latin1', 'Adja-UTF8',
```

Однако, к сожалению, к настоящему времени для многих языков нет существенных корпусов. Это связано с недостаточной государственной или промышленной поддержкой развития лингвистических ресурсов, а индивидуальные усилия являются частичными, трудно определяемыми и, за редким исключением, допускающими повторное использование. Кроме того, некоторые языки не имеют устоявшейся системы письменности и находятся на грани исчезновения.

Работа с текстами различных языков требует использования различных кодировок. Если говорить о текстах на английском языке, то такие тексты обычно используют ASCII-кодировку. Европейские языки обычно используют расширенную латиницу, содержащую, например, такие символы, как ÿ, ò или ê; русский

и украинский языки используют кириллицу и так далее. Обычно для обработки таких текстов в Python необходимо использовать Unicode. В программе можно манипулировать Unicode-строкой подобно обычной строке.

Для того чтобы представить кодировку текста в файле, в Unicode используется декодирование (**decoding**). И, наоборот, чтобы записать текст из Unicode в файл или вывести его на экран, прежде всего, необходимо представить его в подходящей кодировке. Такой процесс называется кодированием (**encoding**).

Для того, чтобы кодировать и декодировать различные кодировки текста, используется модуль **codecs** пакета **nltk**. Модуль **codecs** содержит функции декодирования текста в строку Unicode и, наоборот, кодировки Unicode строки. Функция **codecs.open()** содержит параметр вида кодировки для обозначения способа кодировки открываемого файла.

Например, есть небольшой текстовый файл `polish-lat2.txt`. Имя файла предполагает, что это отрывок польского текста (из Польской Википедии, http://pl.wikipedia.org/wiki/Biblioteka_Pruska) и что файл закодирован как *Latin-2*; кодировка также имеет название *ISO-8859-2*.

Функция **find()** модуля **nltk.data** позволяет находить файл для любого элемента корпуса. Возвращаемое функцией значение может быть использовано в качестве параметра для открытия файла встроенным методом **open()** и методом **codecs.open()** модуля **codecs**. Текст, возвращаемый функцией **codecs.open()** модуля **codecs** для последующего чтения, представлен в Unicode:

```
import nltk
import codecs
path=nltk.data.find('corpora/unicode_samples/polish-lat2.txt')
print(path)
f=open(path)
print(f.read())
f=codecs.open(path, encoding='latin2')
print(f.read())
```

encoding

```
Pruska Biblioteka Państwowa. Jej dawne zbiory znane pod nazwą  
"Berlinka" to skarb kultury i sztuki niemieckiej. Przewiezione przez  
Niemców pod koniec II wojny światowej na Dolny Śląsk, zostały  
odnalezione po 1945 r. na terytorium Polski. Trafiły do Biblioteki  
Jagiellońskiej w Krakowie, obejmują ponad 500 tys. zabytkowych  
archiwaliów, m.in. manuskrypty Goethego, Mozarta, Beethovena, Bacha.
```

```
Pruska Biblioteka Państwowa. Jej dawne zbiory znane pod nazwą  
"Berlinka" to skarb kultury i sztuki niemieckiej. Przewiezione przez  
Niemców pod koniec II wojny światowej na Dolny Śląsk, zostały  
odnalezione po 1945 r. na terytorium Polski. Trafiły do Biblioteki  
Jagiellońskiej w Krakowie, obejmują ponad 500 tys. zabytkowych  
archiwaliów, m.in. manuskrypty Goethego, Mozarta, Beethovena, Bacha.
```

В приведенном выше примере, если файл открывается обычным способом (встроенным методом `open(path)`), то полученная строка текста включает нераспознанные или неверно распознанные символы. В то же время открытие файла с использованием функции декодирования `f=codecs.open(path, encoding='latin2')` позволяет верно прочесть все символы польского языка.

Список параметров, указывающих возможные варианты кодировки текста, можно увидеть на странице <https://docs.python.org/3/library/codecs.html>.

```
import nltk
import codecs
path=nltk.data.find('corpora/unicode_samples/polish-lat2.txt')
print(path)
f=codecs.open(path, encoding='latin2')
for line in f:
    line=line.strip()
    print (line.encode('unicode_escape'))
```

```
b'Pruska Biblioteka Pa\\u0144stwowa. Jej dawne zbiory znane pod nazw\\u0105'  
b'"Berlinka" to skarb kultury i sztuki niemieckiej. Przewiezione przez'  
b'Niemc\\xf3w pod koniec II wojny \\u015bwiatowej na Dolny \\u015al\\u0105sk, zosta\\u0142y'  
b'odnalezione po 1945 r. na terytorium Polski. Trafi\\u0142y do Biblioteki'  
b'Jagiello\\u0144skiej w Krakowie, obejmuj\\u0105 ponad 500 tys. zabytkowych'  
b'archiwali\\xf3w, m.in. manuskrypty Goethego, Mozarta, Beethovena, Bacha.'
```

8.3. Загрузка собственного корпуса

Доступ к каждому корпусу осуществляется с помощью объектов классов **"corpus reader"** модуля **nltk.corpus**. Каждый класс **"corpus reader"** специализирован под определенный формат корпуса. Например, класс **PlaintextCorpusReader** предназначен для обработки корпусов, включающих обычные, неаннотированные текстовые файлы, а класс **BracketParseCorpusReader** — для обработки корпусов, включающих файлы, содержащие скобочно-размеченные деревья синтаксического разбора.

Если необходимо загрузить набор собственных текстов для дальнейшей их обработки методами модуля **nltk.corpus**, используется класс **PlaintextCorpusReader()** модуля **nltk.corpus**. Конструктор класса имеет два параметра: первый параметр представляет собой строку, указывающую директорию, из которой будут закачиваться текстовые файлы корпуса; второй параметр представляет собой список файлов, подобно ['a.txt', 'test/b.txt'] или регулярное выражение, подобное '[abc]/.*\.txt'.

Например, если текстовые файлы расположены в каталоге d:/ФайлыPython/KZ_corpus, то фрагмент кода, позволяющего загрузить все файлы каталога будет выглядеть следующим образом:

```
from nltk.corpus import PlaintextCorpusReader
wordlist=PlaintextCorpusReader("d:/ФайлыPython", "1_zakon_20.07.2018_ru_
_raw.txt")
print(wordlist.raw())
wordlist=PlaintextCorpusReader("d:/ФайлыPython/KZ_corpus", ".*")
print (wordlist.raw())
print(wordlist.fileids())
```

8.4. Работа с размеченными корпусами в NLTK

Библиотека NLTK содержит много лингвистически аннотированных текстовых корпусов, включающих *POS-tagging*, *named entities*, *syntactic structures*, *semantic roles* и другие виды аннотации. NLTK предлагает удобный путь доступа к этим корпусам и имеет пакеты данных, содержащие образцы корпусов, свобод-

но загружаемых⁹ и используемых для преподавания и исследований¹⁰.

В морфологически аннотированные корпуса добавлена POS-разметка. Например, если открыть **Brown Corpus** с помощью текстового редактора, то можно увидеть следующий текст, в котором после каждого токена стоит слеш и POS-тег:

The/at Fulton/np-tl County/nn-tl Grand/jj-tl Jury/nn-tl said/vbd Friday/nr an/at investigation/nn of/in Atlanta's/np\$ recent/jj primary/nn election/nn produced/vbd / no/at evidence/nn "/" that/cs any/dti irregularities/nns took/vbd place/nn ./.

Другие корпуса могут применять иной формат разметки. Например, горизонтальную разметку, использующую символ подчеркивания:

The_AT Fulton_NP1 County_NN1 Grand_JJ Jury_NN1 said_VVD Friday_NPD1 an_AT1 investigation_NN1 of_IO Atlanta_NP1 's_GE recent_JJ primary_JJ election_NN1 produced_VVD no_AT evidence_NN1 that_CST any_DD irregularities_NN2 took_VVD place_NN1 ._YSTP

Библиотека `nltk.corpus` предоставляет стандартный интерфейс работы с размеченными корпусами, не зависящий от форматов POS-разметки, используемых различными файлами. Если корпус является аннотированным, то **nltk.corpus**-интерфейс имеет метод **tagged_words()**:

```
import nltk
print(nltk.corpus.brown.tagged_words())
print(nltk.corpus.brown.tagged_words(tagset='universal'))
print("Корпус nps_chat")
print(nltk.corpus.nps_chat.tagged_words())
```

```
C:\Users\Nina\PycharmProjects\project_1\venv\Scripts\pytl
[('The', 'AT'), ('Fulton', 'NP-TL'), ...]
[('The', 'DET'), ('Fulton', 'NOUN'), ...]
Корпус nps_chat
[('now', 'RB'), ('im', 'PRP'), ('left', 'VBD'), ...]
```

⁹ http://www.nltk.org/nltk_data/

¹⁰ <http://www.nltk.org/data>

POS-tag метки традиционно, с момента публикации Браунского корпуса, отображаются большими буквами. Однако не все корпуса используют одно и то же множество морфологических меток (tagset). Для того чтобы избежать возможные сложности отображения этих меток, можно использовать встроенное отображение универсальных POS-tag меток “**Universal Tagset**”, фрагмент которого показан в табл. 8.1.

Таблица 8.1 – Множество универсальных POS-меток (Universal Part-of-Speech Tagset)

тег	значение	Пример
ADJ	прилагательное	<i>new, good, high, special, big, local</i>
ADV	наречие	<i>really, already, still, early, now</i>
CONJ	частица	<i>and, or, but, if, while, although</i>
DET	артикли, определители	<i>the, a, some, most, every, no, which</i>
NOUN	существительное	<i>year, home, costs, time, Africa</i>
NUM	числительное	<i>twenty-four, fourth, 1991, 14:24</i>
PRT	предлог	<i>at, on, out, over per, that, up, with</i>
PRON	местоимение	<i>he, their, her, its, my, I, us</i>
VERB	глагол	<i>is, say, told, given, playing, would</i>
.	знак пунктуации	<i>. , ; !</i>
X	другое	<i>ersatz, esprit, dunno, gr8, univeristy</i>

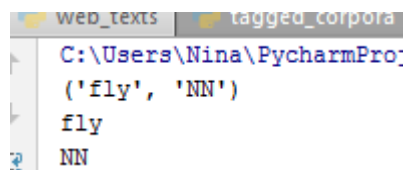
Обычно размеченные корпуса также обладают методом **tagged_sents()**, который выделяет размеченные слова каждого предложения в отдельный список.

В NLTK размеченный токен представляется в виде кортежа, состоящего из токена и тега. С помощью функции **str2tuple()** модуля **nltk.tag** можно создать такой кортеж с помощью стандартной строки, представляющей размеченный токен:

```
import nltk
tagged_token=nltk.tag.str2tuple('fly/NN')
print(tagged_token)
```



```
print(tagged_token[0])
print(tagged_token[1])
```



```
web_texts tagged_corpus
C:\Users\Nina\PycharmPro:
('fly', 'NN')
fly
NN
```

Можно создать список таких кортежей (токен, тег), обрабатывая текст размеченного корпуса:

```
sent=""" The/AT Fulton/NP1 County/NN1 Grand/JJ Jury/NN1 said/VVD Friday/NPD1 an/AT1 investigation/NN1 of/IO Atlanta/NP1 's/GE recent/JJ primary/JJ election/NN1 produced/VVD no/AT evidence/NN1 that/CST any/DD irregularities/NN2 took/VVD place/NN1 ./YSTP """
```

```
list_tags=[nltk.tag.str2tuple(t) for t in sent.split()]
print(list_tags)
```

```
[('The', 'AT'), ('Fulton', 'NP1'), ('County', 'NN1'), ('Grand', 'JJ'), ('Jury', 'NN1'), ('said', 'VVD'), ('Friday', 'NPD1'),
```

Наиболее удобным для хранения пар (**word, tag**) является структура словарей.

Контрольные вопросы и практические задания по теме «Работа с корпусами в nltk»

1. На файлах корпуса Проекта Гутенберга покажите, что средняя длина слова каждого анализируемого файла корпуса равна чеырем символам, а средняя длина предложения в каждом файле разная:

а) Известно, что средняя длина слова английского языка равна трем символам. Как это сочетается с проведенным исследованием?

б) В чем измеряется длина предложения?

2. Определите количество файлов в корпусе webtext.

3. Для файлов pirates.txt и wine.txt корпуса webtext определите:

- количество токенов;
- количество уникальных токенов;

- лексическое разнообразие;
- среднюю длину предложения;
- первые 50 слов;
- первые 50 предложений;
- первые 50 символов.

Выведите текст корпусов на экран.

4. Выведите на экран файл описания корпуса Брауна (файл README), категории данного корпуса и файлы одной любой категории.
5. Определите, к каким общим категориям относятся файлы 'training/9971' и 'test/27577' корпуса reuters.
6. Какие файлы относятся к категориям 'castor-oil', 'cocoa', 'coconut', 'coconut-oil', 'coffee', 'copper' корпуса reuters?
7. Определите первые десять слов файлов 'training/9971' и 'test/27577' корпуса reuters.
8. Определите список слов категории 'copper' корпуса reuters.
9. Найдите в корпусе Universal Declaration of Human Rights все файлы, содержащие текст декларации на украинском языке. Раскодируйте тексты в формат Unicode и выведите тексты на экран.
10. Загрузите файлы архива gu.hug в качестве собственного корпуса. Выведите список файлов корпуса, количество слов корпуса, количество уникальных токенов в корпусе, текст нескольких любых файлов корпуса.
11. Сравните внутреннюю разметку и встроенное отображение POS-меток корпусов conll2000 и treebank.
12. Получите списки предложений тагелированных слов корпусов brown и conll2000.

Тема 9. ПРОСТЫЕ СТАТИСТИЧЕСКИЕ ИССЛЕДОВАНИЯ КОРПУСОВ

9.1. Создание объект nltk.Text.

9.2. Использование пакета Matplotlib для построения графиков и диаграмм.

9.3. Построение частотного распределения слов в корпусе.

9.4. Коллокации и биграммы (collocations and bigrams) в тексте.

Контрольные вопросы и практические задания по теме «Простые статистические исследования корпусов».

9.1. Создание объекта nltk.Text

Для того чтобы осуществлять различные статистические исследования текста, необходимо представить его в виде **nltk-текста** или в виде объекта класса nltk.Text, используя в качестве параметра список слов (токенов).

Например, можно создать **nltk-текст**, используя список слов файла произведения У. Шекспира “Макбет” корпуса gutenber:

```
shek=nltk.Text(gutenberg.words(fileids="shakespeare-macbeth.txt"))
```

Существует много способов исследовать текст, представленный в виде **nltk-текста**. Использование метода **concordance** объекта **nltk.текст** позволяет увидеть варианты появления данного слова во всевозможных контекстах. Слово, для которого определятся контекст, используется в качестве параметра метода

```
from nltk.corpus import gutenber
import nltk
shek=nltk.Text(gutenberg.words(fileids="shakespeare-macbeth.txt"))
print(type(shek))
shek.concordance("Macbeth")
```

```
<class 'nltk.text.Text'>
```

```
Displaying 25 of 25 matches:
```

```
Macbeth by William Shakespeare 1603 ] Actus
```

```
on the Heath 3 . There to meet with Macbeth 1 . I come , Gray - Malkin All . Pa
: but all ' s too weake : For braue Macbeth ( well hee deserues that Name ) Dis
Dismay ' d not this our Captaines , Macbeth and Banquoh ? Cap . Yes , as Sparro
h , And with his former Title greet Macbeth Rosse . Ile see it done King . What
ne King . What he hath lost , Noble Macbeth hath wonne . Exeunt . Scena Tertia
within 3 . A Drumme . A Drumme . Macbeth doth come All . The wounded Sisters
```

Метод **similar()** объекта `nlk.Text` позволяет проанализировать список слов, встречающихся в таком же контексте, как и слово, являющееся параметром данного метода.

Метод **common_contexts ()**, параметрами которого являются два слова, позволяет увидеть общий (одинаковый) контекст, в котором эти два слова встречаются:

```
from nltk.corpus import gutenberg
import nltk
shek=nlk.Text(gutenberg.words(fileids="shakespeare-macbeth.txt"))
shek.count("Macbeth")
print()
shek.similar("Macbeth")
print()
shek.common_contexts("Macbeth", "king")
shek.common_contexts(("Macbeth", "lady")
```

61

```
that king lenox what rosse lady macduff of by in be vpon with a which
not were cawdor against vs
```

```
haile_that enter_lenox
enter_how enter_lady
```

Метод **dispersion_plot ()** позволяет увидеть распределение списка заданных в качестве параметров слов в тексте. Например, можно увидеть, как часто данные слова появляются в начале текста, в его средней части и т. д.

```
shek.dispersion_plot(["Macbeth", "King", "Lady", "Enter", "Witches"])
```

На этом графике распределения лексики (Lexical Dispersion Plot) по оси ординат откладываются слова, а по оси абсцисс откладывается смещение в словах относительно начала текста. Например, на графике распределения лексики, заданной в качестве списка запрашиваемых слов, который показан на рис. 9.1, мож-

но увидеть, что слово “Witches” встречается в начале трагедии “Макбет”, а затем еще несколько раз в начале второй половины текста.

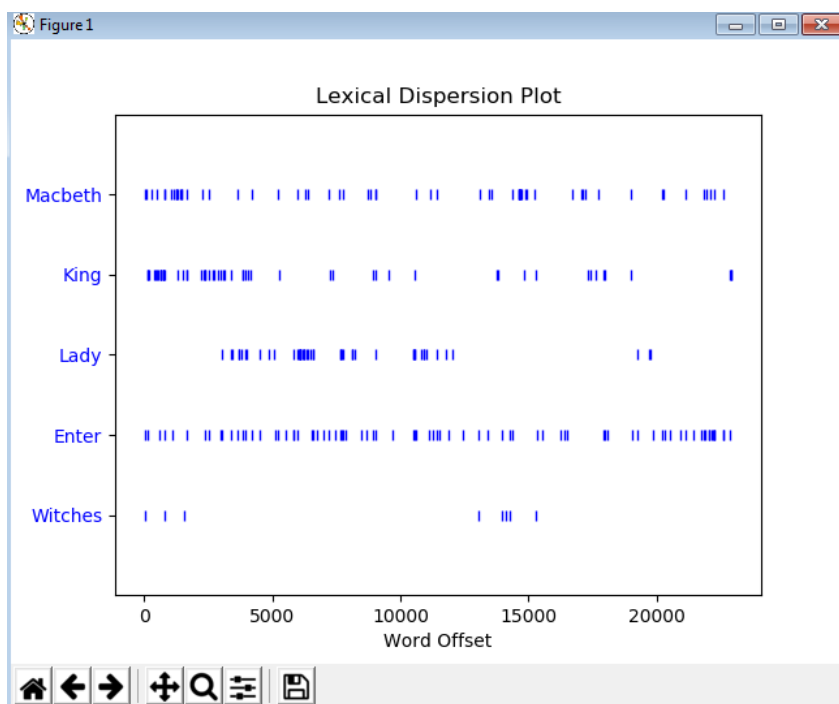


Рисунок 9.1 – График распределения слов "Macbeth", "King", "Lady", "Enter", "Witches" в nltk Text файла shakespeare-macbeth.txt библиотеки Гутенберга

9.2. Использование пакета Matplotlib для построения графиков и диаграмм

Для того чтобы иметь возможность строить различные графики и диаграммы, необходимо установить пакеты **NumPy** и **Matplotlib**. Библиотека **Matplotlib** используется для визуализации данных, для создания диаграмм и графиков. Данная библиотека организована иерархически. Чтобы нарисовать простую гистограмму, достаточно использовать функцию **hist(arr)** интерфейса высшего уровня **pyplot** библиотеки **matplotlib**:

```
matplotlib.pyplot.hist(arr).
```

Существует фактически стандарт вызова **pyplot**, с использованием алиаса (alias) **plt**:

```
import matplotlib.pyplot as plt
```

или:

```
from matplotlib import pyplot as plt
```

Для того чтобы результат рисования отразился на экране, можно воспользоваться командой **plt.show()**.

Например, можно получить распределение токенов (слов) по их длинам (рис. 9.2):

```
from matplotlib import pyplot as plt
from nltk.tokenize import word_tokenize
words=word_tokenize("This is a pretty cool tool!")
word_lengths=[len(w) for w in words]
plt.hist(word_lengths)
plt.show()
```

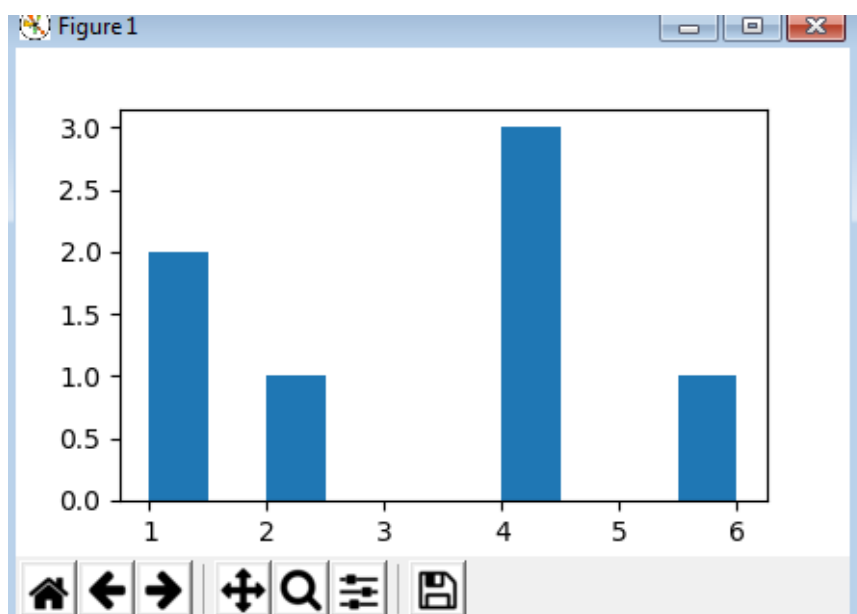


Рисунок 9.2 – График распределения слов по их длинам

9.3. Построение частотного распределения слов в корпусе

Работая с объектом `nltk.Text`, можно сразу получить множество уникальных токенов данного текста и посчитать число появлений того или иного токена в тексте:

```
print(sorted(set(shek)))
print (shek.count("Macbeth"))
```

```
[ '!', '€', '"', '(', ')', ',', '-', '.', '1', '1603', '2', '3', ':', ';', '?', 'A', 'Accents', '
61
```

В начале списка отсортированных уникальных токенов располагаются знаки пунктуации, цифры, а потом слова с большой буквы, после слова с маленькой буквы.

Во многих задачах NLP необходимо получить частотное распределение слов в тексте. Например, для определения слов, которые являются наиболее информативными для текстов определенного жанра или определенной тематики, можно определить частоты появления слов в данном тексте, т. е. построить частотное распределение. Частотное распределение (**frequency distributions**) указывает на частоту, с которой в тексте встречается каждое из слов. Такой частотный список называют распределением, так как общее количество слов распределяется между словарными статьями (оригинальными словами) в тексте.

В NLTK реализован отдельный класс **FreqDist** в модуле *nltk.probability*. Объект данного класса представляет собой словарь, к которому можно применить основные методы словаря:

```
from nltk.corpus import gutenberg
import nltk
from matplotlib import pyplot as plt
print(gutenberg.fileids())
shakespeare=gutenberg.raw(fileids='shakespeare-macbeth.txt')
shek=nltk.Text(gutenberg.words(fileids="shakespeare-macbeth.txt"))
fdist1=nltk.FreqDist(shek)
print(type(fdist1))
print(len(fdist1)) #количество уникальных слов
print(list(fdist1.items())[:10]) #первый десять кортежей ("слово", частота)
print(sorted(list(fdist1.values()),reverse=True)[:10])
print(sorted(list(fdist1.keys()),reverse=True)[:10])
print(fdist1.keys())
for key in list(fdist1.keys())[:10]:
    print ("%s ->%s"%(key, fdist1[key]))
```

```

<class 'nlTK.probability.FreqDist'>
4017
[(' ', 4), ('The', 118), ('Tragedie', 1), ('of', 315), ('Macbeth', 61), ('by', 36), ('William', 1),
[1962, 1235, 637, 531, 477, 376, 333, 315, 311, 241]
['youths', 'youth', 'yours', 'your', 'young', 'youl', 'you', 'yong', 'yoake', 'ynch']
dict_keys([' ', 'The', 'Tragedie', 'of', 'Macbeth', 'by', 'William', 'Shakespeare', '1603', ']', 'Ac'
[ ->4
The ->118
Tragedie ->1
of ->315
Macbeth ->61
by ->36
William ->1
Shakespeare ->1
1603 ->1
] ->4

```

Можно построить график частотного распределения слов (ранг-частота), который полностью соответствует Закону Ципфа (рис. 9.3):

```
fdist1.plot(50,cumulative=False)
```

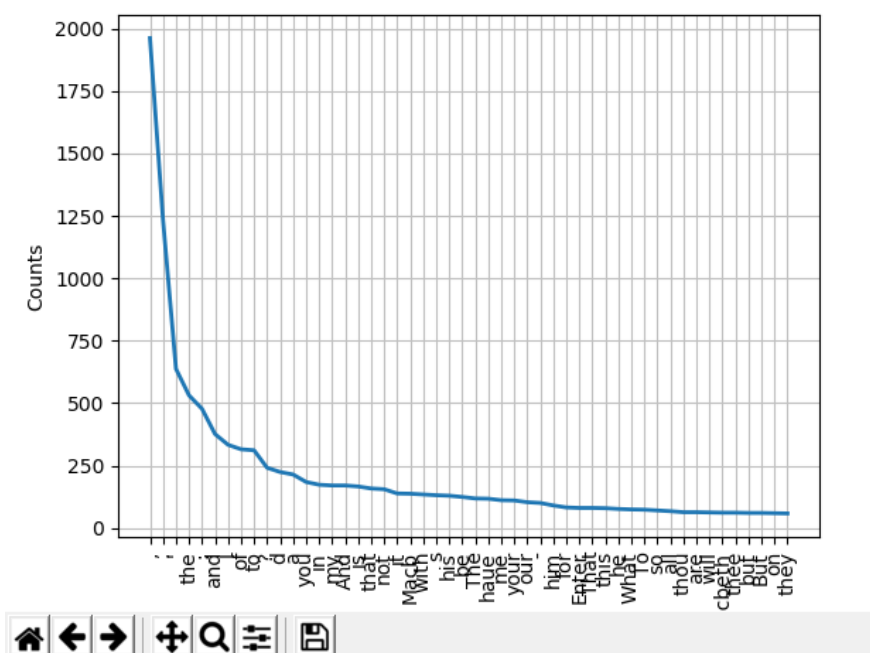


Рисунок 9.3 – График частотного распределения пятидесяти наиболее частых слов поэмы Шекспира «Макбет»

Можно получить кумулятивный график частоты (накапливающий или суммирующий частоты) наиболее частотных слов текста (рис. 9.4):

```
fdist1.plot(30,cumulative=True)
```

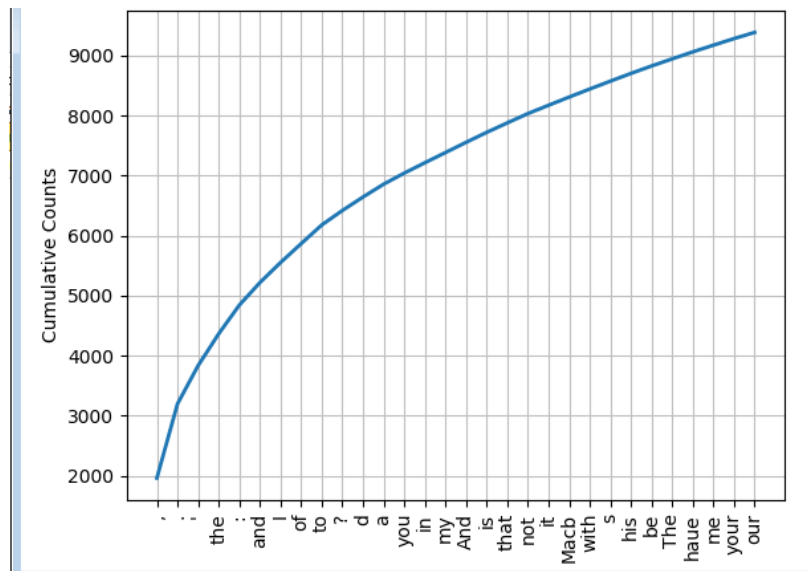



Рисунок 9.4 – Кумулятивный график частотного распределения тридцати наиболее частых слов поэмы Шекспира «Макбет»

Используя метод **hapaxes()** объекта класса **FreqDist**, можно получить список слов, встречающихся в данном nltk-тексте только один раз:

```
print(fdlist1.hapaxes())
```

```
['Tragedie', 'William', 'Shakespeare', '1603', 'Primus', 'Scoena', 'Raine', 'Hurley', 'burley', 'Battaile', 'Gray', 'Malkin',
```

Используя небольшой фрагмент кода, можно получить список слов, встречающихся в тексте больше 7 раз и имеющих длину больше 7 символов:

```
print ("список слов, длиной больше 7 символов и встречающиеся больше 7 раз")
```

```
fre_long=[w for w in set(shek) if len(w)>7 and fdlist1[w]>7]
```

```
print (sorted(fre_long))
```

```
список слов, длиной больше 7 символов и встречающиеся больше 7 раз
```

```
['Businesse', 'Children', 'Dunsinane', 'Highnesse', 'Macduffe', 'Malcolme', 'Scotland']
```

Кроме того, используя объект класса **FreqDist**, можно получить распределение длин слов в тексте:

```
fdist2=nltk.FreqDist([len(w) for w in shek])
```

В табл. 9.1 показаны некоторые дополнительные возможности работы с объектом частотного распределения (FreqDist) текста nltk.

Таблица 9.1 – дополнительные возможности работы с объектом частотного распределения (FreqDist) текста nltk

fdist=FreqDist(текст_nltk)	Создает частотное распределение текста nltk
fdist['Word']	Определяет число появлений слова 'Word' в тексте
fdist.freq('Word')	Определяет частоту появления слова 'Word' в тексте
fdist.N()	Определяет количество уникальных токенов в тексте
for sample in fdist:	Осуществляет итерацию по всем уникальным токенам в порядке убывания частоты
fdist.max()	Определяет наиболее частотный токен текста
fdist.tabulate()	Создает табличное представление частотного распределения слов в тексте
fdist.plot()	Строит диаграмму частотного распределения
fdist.plot(cumulative=True)	Строит кумулятивную диаграмму (диаграмму с накоплением) частотного распределения слов
fdist1<fdist2	Определяет слова, частота появления которых в fdist1 меньше, чем в распределении fdist2

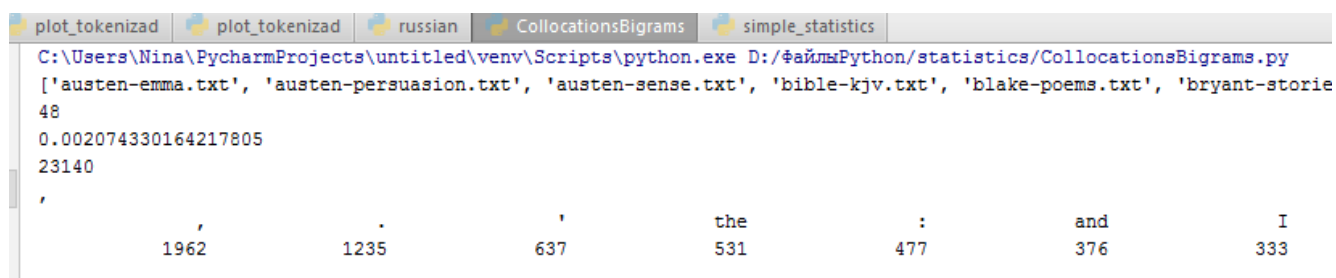
Ниже приведен пример использования некоторых, приведенных в табл. 9.1. функций и методов работы с частотным распределением слов в корпусе:

```
from nltk.corpus import gutenber
import nltk
from matplotlib import pyplot as plt
print(gutenber.fileids())
shakespeare=gutenber.raw(fileids='shakespeare-macbeth.txt')
```

```

shek=nlk.Text(gutenberg.words(fileids="shakespeare-macbeth.txt"))
fdist1=nlk.FreqDist(shek)
print(fdist1['Lady'])
print(fdist1.freq('Lady'))
print(fdist1.N())
print(fdist1.max())
fdist1.tabulate()

```



9.4. Коллокации и биграммы (collocations and bigrams) в тексте

Очень часто при частотном анализе тестов используется анализ n-граммы (Bigram, Trigram). **Биграмма** представляет собой последовательность из пары слов в тексте. Можно получить всевозможные биграммы на списке слов или токенов, используя метод **bigrams()**.

Коллокации представляют собой последовательности слов, которые проявляются вместе чаще, чем они проявлялись бы, если бы появлялись в тексте независимо друг от друга. В некотором приближении можно сказать, что двухсловные коллокации – это часто встречающиеся биграммы. То есть практически, необходимо определить биграммы, которые появляются в тексте чаще, чем ожидается исходя из частоты появления составляющих их слов. Метод **collocations()** объекта **nlk_текст** позволяет получить список коллокаций в заданном тексте:

```

print("Bigramm and Collocations")
print(list(nltk.bigrams(['more', 'than', 'your', 'knowledge', 'is'])))
shek.collocations()

```

```

> Bigramm and Collocations
> [('more', 'than'), ('than', 'your'), ('your', 'knowledge'), ('knowledge', 'is')]
> Enter Macbeth; Scena Secunda; three Witches; good Lord; thou art; Fire
> burne; Scena Prima; worthy Thane; ten thousand; weyward Sisters; mine
> owne; Cauldron bubble; haue done; Old man; Scena Quinta; thy face;
> Enter Malcolme; Scena Quarta; Lord Macb; Scena Tertia

```

Контрольные вопросы и практические задания по теме «Простые статистические исследования корпусов»

1. Для осуществления статистических исследований текста трагедии У. Шекс-пира “Гамлет” представьте в виде текста nltk файл shakespeare-hamlet.txt библиотеки gutenberг.
2. Просмотрите контекст появления слова “Hamlet” в данной трагедии. Сколь-ко раз встречается данное слово?
3. Найдите в данном тексте общий контекст слов "Hamlet" и " Horatio " .
4. Постройте график распределения слов "Hamlet", " Horatio ", “Ghost “, и “Polonius” в тексте драмы.
5. Определите число уникальных токенов в корпусе и отобразите их список.
6. Постройте график частотного распределения слов в корпусе и кумулятивный график частотного распределения слов в корпусе.
7. Покажите слова, которые встречаются в корпусе только один раз.
8. Составьте два списка слов: 1) длиной больше 7 символов, частота появления которых в корпусе больше семи; 2) длиной меньше 3 символов, встречающихся в тексте меньше трех раз. Сравните списки. Как вы думаете, какой из списков передает больше информации о смысле текста и почему?
9. Создайте функцию, позволяющую определять лексическое богатство текста (количество слов в тексте, деленное на количество уникальных слов).
10. Создайте функцию, определяющую вероятность появления слова, переданного в функцию в качестве параметра в корпусе, размер которого (количество слов) также передается в функцию в качестве параметра.
11. Определите коллокации данного корпуса. Объясните, что такое коллокации.

12. Составьте список слов текста, имеющих длину больше 12 символов.
13. Получите распределение длин слов в корпусе.
14. Определите, какая длина слова встречается в корпусе наиболее часто.
15. Постройте диаграмму распределения частот длин слов.
16. Определите, какие длины слов встречаются в тексте.
17. Выведите на экран имеющиеся длины слов, с указанием частотности их появления.
18. Постройте диаграмму распределения частот длин слов.

СПИСОК ЛИТЕРАТУРЫ

1. An algorithm for suffix stripping. Program 14.3. – 1980. – p. 130–137 [Электронный ресурс]. – Режим доступа : <https://tartarus.org/martin/PorterStemmer/>
2. Bird S. Natural Language Processing with Python / S. Bird, E. Klein, E. Loper. – O'Reilly, 2009. – 502 p.
3. Lane H. Natural Language Processing in Action. Understanding, analyzing, and generating text with Python / H. Lane, C. Howard, H. M. Naepe. – Manning Publications, 2019. – 544 p.
4. Paice C. D. Another stemmer / C. D. Paice // SIGIR Forum, 1990. – 24(3). – p. 56–61.
5. Srinivasa-Desikan B. Natural Language Processing and Computational Linguistics: A practical guide to text analysis with Python, Genim, spaCy, and Keras / B. Srinivasa-Desikan. – Packt Publishing, 2018. – 306 p.
6. Любанович Б. Простой Python. Современный стиль программирования / Б. Любанович. – СПб. : Питер, 2016. – 480 с.
7. Прохоренок Н. А. Python. Самое необходимое / Н. А. Прохоренок. – СПб. : БХВ-Петербург, 2011. – 416 с.
8. Бенгфорт Б. Прикладной анализ текстовых данных на Python. Машинное обучение и создание приложений обработки естественного языка / Б. Бенгфорт, Р. Билбро, Т. Охеда. – СПб. : Питер, 2019. – 368 с.
9. Копотев М. Введение в корпусную лингвистику / М. Копотев. – Прага : Animedia Company, 2014. – 230 с.

Навчальне видання

ХАЙРОВА Ніна Феліксівна
МАМИРБАЄВ Оркен Жумажанович
ПЕТРАСОВА Світлана Валентинівна
МУХСІНА Куралай Женисбеківна

**СУЧАСНІ ТЕХНОЛОГІЇ ОБРОБКИ ТЕКСТОВИХ ДАНИХ
НА БАЗІ ПАКЕТА NLTK PYTHON**

Навчальний посібник
з курсу «Сучасні технології обробки текстових даних»
для студентів спеціальності «Прикладна та комп'ютерна лінгвістика»

Відповідальний за випуск проф. *Шаронова Н. В.*
Роботу до видання рекомендував проф. *Дмитрієнко В. Д.*

Редактор *Л. Л. Яковлева*

Видання російською мовою

План 2020 р., поз. 71

Видавничий центр НТУ «ХП».
Свідоцтво про державну реєстрацію ДК № 5478 від 21.08.2017 р.
61002, Харків, вул. Кирпичова, 2

Підп. до друку 18.09.2020. Формат 60×84 1/16. Папір офсетний.
Друк цифровий. Гарнітура Times New Roman. Ум. друк. арк. 6,7.
Наклад 50 прим. Зам. № 11/09/20. Ціна договірна.

Видавець: ТОВ "В СПРАВІ"
Свідоцтво серія ДК № 4845 від 06.02.2015 р.
м. Харків, вул. Жон Мироносиць, 10, оф. 6,
тел. +38(057)714-06-74, +38(050)976-32-87
copy@vlavke.com

Виготовлювач: ФОП Панов А.М.
Свідоцтво серії ДК № 4847 від 06.02.2015 р.
м. Харків, вул. Жон Мироносиць, 10, оф. 6,
тел. +38(050)976-32-87